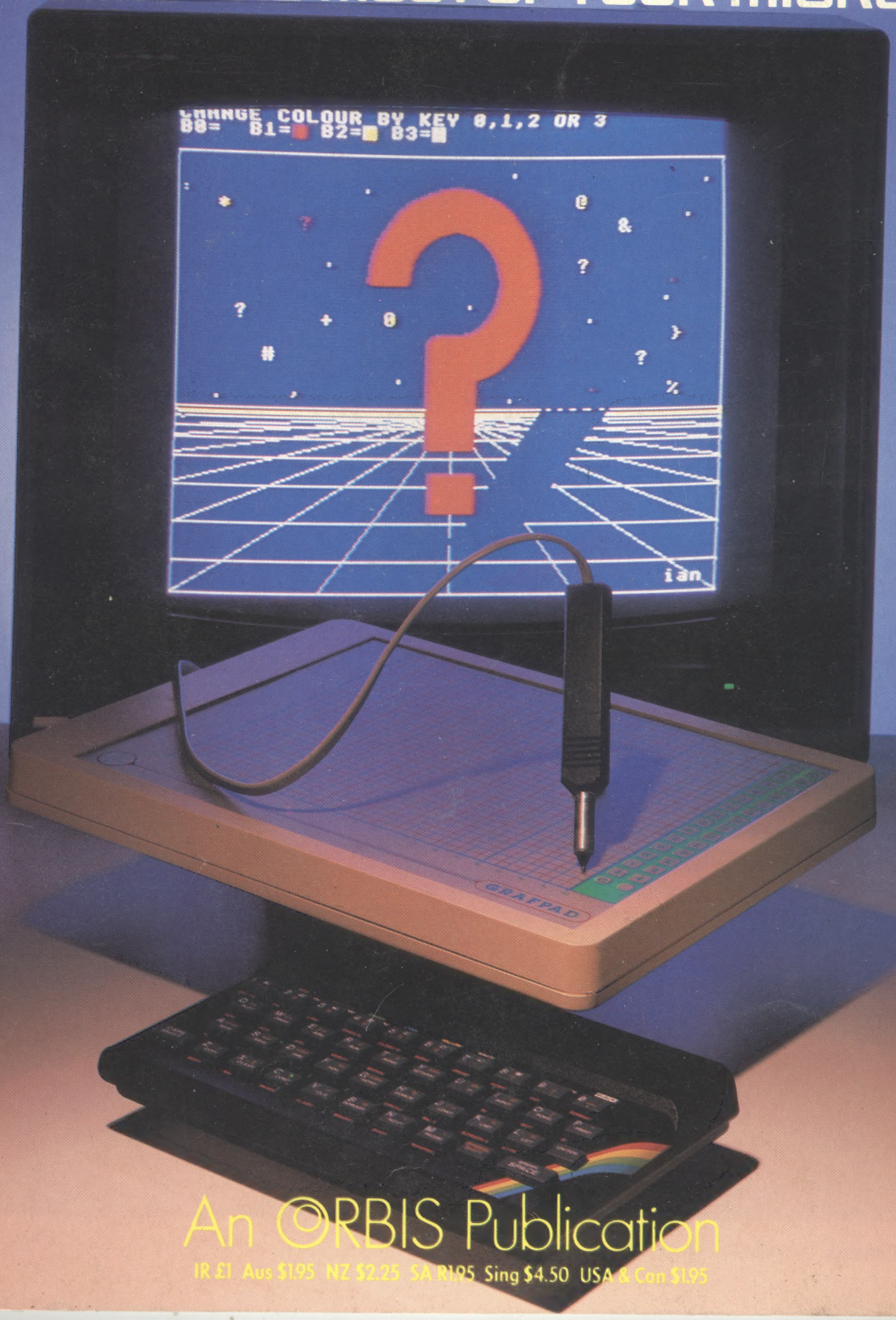
ISSN 0265-2919

80p 9

THE HOME COMPUTER ADVANCED COURSE

MAKING THE MOST OF YOUR MICRO



An ORBIS Publication

IR £1 Aus $1.95 NZ $2.25 SA R1.95 Sing $4.50 USA & Can $1.95

CONTENTS

APPLICATION

SURVIVAL OF THE FITTEST How the home computer world came to be dominated by two microprocessor designs 161

HARDWARE

SCRIBBLE PAD We take a look at the Grafpad, a graphics tablet for home micros 169

SOFTWARE

TAKING STOCK Our series on business packages continues with an analysis of a stock recording system 172

COMPUTER SCIENCE

ALTERNATIVE PATHS We introduce two new logic gates – NAND and NOR 166

PROGRAMMING PROJECTS

DEGREES OF PRECISION We look at how to use trig functions in BASIC programs 174

JARGON

FROM BASIC TO BISTABLE A weekly glossary of computing terms 168

MACHINE CODE

STARTING FLAG We look at the function of the flags in the processor status register 176

PROFILE

GETTING IT TAPED Virgin Games is a new company that sees connections between marketing records and computer games 180

WORKSHOP

HALF MEASURE We show you how to build a half adder circuit 164

Next Week

- We look at the two enhanced Atari home micros: the 600XL and 800XL and their exciting range of peripherals.
- Operating systems are the housekeepers of every computer. We examine the vital function they perform.

COMING VERY SOON

FREE

MACHINE CODE MONITOR AND ASSEMBLER PROGRAM ON CASSETTE

Written specially for Home Computer Advanced Course readers

Editor Max Phillips; **Art Director** David Whelan; **Production Editor** Catherine Cardwell; **Staff Writer** Brian Morris; **Picture Editor** Claudia Zeff; **Designers** Hazel Bennington, Julian Dorr; **Sub Editor** Robert Pickering; **Art Assistant** Liz Dixon; **Editorial Assistant** Stephen Malone; **Contributors** Lisa Kelly, Steven Colwill, Geoff Bains, Tony Harrington, Richard Pawson; **Consultant Editor** Gareth Jefferson; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd; Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editor** Chris Cooper; **Production Co-ordinator** Ian Paton; **Circulation Director** David Breed; **Marketing Director** Michael Joyce; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 85 Charlotte Street, London W1P 1LB; **© APSIF Copenhagen 1984; © Orbis Publishing Ltd 1984: Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Artisan Press Ltd, Leicester**

HOME COMPUTER ADVANCED COURSE – Price UK 80p IR £1.00 AUS $1.95 NZ $2.25 SA R1.95 SINGAPORE $4.50 USA and CANADA $1.95

How to obtain your copies of HOME COMPUTER ADVANCED COURSE – Copies are obtainable by placing a regular order at your newsagent, or by taking out a subscription. Subscription rates: for six months (26 issues) £23.80; for one year (52 issues) £47.60. Send your order and remittance to Punch Subscription Services, Watling Street, Bletchley, Milton Keynes, Bucks MK2 2BW, being sure to state the number of the first issue required.

Back Numbers UK and Eire – Back numbers are obtainable from your newsagent or from HOME COMPUTER ADVANCED COURSE. Back numbers, Orbis Publishing Limited, 20/22 Bedfordbury, LONDON WC2N 4BT at cover price. AUSTRALIA: Back numbers are obtainable from HOME COMPUTER ADVANCED COURSE. Back numbers, Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G Melbourne, Vic 3001. SOUTH AFRICA, NEW ZEALAND, EUROPE & MALTA: Back numbers are available at cover price from your newsagent. In case of difficulty write to the address in your country given for binders. South African readers should add sales tax.

How to obtain binders for HOME COMPUTER ADVANCED COURSE – UK and Eire: Please send £3.95 per binder if you do not wish to take advantage of our special offer detailed in Issues 5, 6 and 7. EUROPE: Write with remittance of £5.00 per binder (incl. p&p) payable to Orbis Publishing Limited, 20/22 Bedfordbury, LONDON WC2N 4BT. MALTA: Binders are obtainable through your local newsagent price £3.95. In case of difficulty write to HOME COMPUTER ADVANCED COURSE BINDERS, Miller (Malta) Ltd, M.A. Vassalli Street, Valletta, Malta. AUSTRALIA: For details of how to obtain your binders see inserts in early issues or write to HOME COMPUTER ADVANCED COURSE BINDERS, First Post Pty Ltd, 23 Chandos Street, St. Leonards, NSW 2065. The binders supplied are those illustrated in the magazine. NEW ZEALAND: Binders are available through your local newsagent or from HOME COMPUTER ADVANCED COURSE BINDERS, Gordon & Gotch (NZ) Ltd, PO Box 1595, Wellington. SOUTH AFRICA: Binders are available through any branch of Central Newsagency. In case of difficulty write to HOME COMPUTER ADVANCED COURSE BINDERS, Intermag, PO Box 57394, Springfield 2137.

Note – Binders and back numbers are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK market only and may not necessarily be identical to binders produced for sale outside the UK. Binders and issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.

COVER PHOTOGRAPHY BY IAN McKINNELL

SURVIVAL OF THE FITTEST

There are millions of microprocessors in use throughout the world, doing jobs as diverse as controlling microwave ovens and video recorders, to powering the familiar home computer. And yet, despite the huge numbers involved and the diversity of applications, the market is dominated by just two designs: the Z80 and the 6502.

Computers on a chip came about almost by accident. In 1972, the chip manufacturer Intel was asked by Datapoint to develop a chip to replace the large number of TTL (transistor-transistor-logic) chips needed in the computer terminals of the time. The product they came up with was called the 8008. It was capable of processing data eight bits at a time, and would have made an ideal 'logic replacement' for use in Datapoint's terminals but for one drawback – it operated too slowly. Although Datapoint decided not to use it, the 8008's potential as a general-purpose computer CPU was soon spotted by engineers and hobbyists, and thus the affordable, desk-top computer was born.

The limitations of the 8008 in terms of speed and power soon became apparent, however, and so Intel set about designing a replacement. The chip they developed, the 8080, rapidly established itself as the dominant force in the market.

At about the same time as Intel announced the 8080, their competitors Motorola launched an eight-bit microprocessor called the 6800. The design philosophies behind the 8080 and the 6800 differed considerably, but they were equally powerful and suitable for use as the basis of a microcomputer design.

Although the 8080 and the 6800 were equally efficient, an accident of history paved the way for the phenomenal success of a third chip, the Z80. In 1974, Gary Kildall, now President of Digital Research, produced a disk operating system for Intel called CP/M. This allowed 8080-based computers to be used with the recently introduced Shugart floppy disk drives. Kildall's operating system was rejected by Intel, who thought that existing software was sufficient to use with the standard mainframe computer systems of the time.

However, smaller computers were becoming increasingly popular and CP/M greatly facilitated file-handling on these systems. This fact ensured the market dominance of the 8080 for many years, and cast the Motorola 6800 into relative obscurity. Various attempts were made to provide comparable disk operating systems for the 6800, but the impetus had gone to the 8080, leaving the 6800 in the cold.

The Vital Choice
Most home micros make the choice between a 6502 processor (as in the BBC Micro) or a Z80 (for example, in the Spectrum). The Dragon, one of the few to use other chips, has a 6809

As the market for microprocessor-based products grew, chip manufacturers scrambled to come up with new designs, but always had to contend with the reluctance of the market to accept anything new unless it offered significant advantages. Investments in hardware design and software production also inhibited the adoption of any new, incompatible microprocessor.

A stroke of genius gave an unexpected break to a new chip design — the Z80. Zilog, a team of design engineers who had previously worked on the 8080 for Intel, realised that the instruction set could be extended. In other words, not all of the possible combinations of ones and zeros that could be recognised by the 8080 as instructions had been exploited. By using binary combinations not used by the Intel chip, Zilog were able to design a microprocessor that would perform identically to the 8080 when supplied with 8080 instructions, but that could offer a considerable improvement in performance. They were thus able to create a chip that used software written for the 8080.

Chip Chart

Microprocessors have evolved from two main sources: those stemming from the original Intel microprocessors and those from Motorola's rival 6800 chip. This chart shows the way the chips developed, as well as some of the machines in which they have been used. Many of the less well-known chips appear in the less popular micros. The Apple III is perhaps the only business machine to use a 6502 processor. The Olivetti M20 is the only general-purpose micro to use a Z8000. In both cases, the unusual choice of microprocessor and its consequent lack of software has inhibited the success of the machine. Some immensely successful machines, like the IBM PC, have the effect of making a chip very popular

Intel

4004

8008

4004 and 8008: Designed to replace large numbers of TTL integrated circuits, the 4004 was a very simple chip that could only handle data in four-bit groups. Intel quickly progressed to eight-bit processing with the 8008. Hobbyists and engineers recognised the potential and began building their own 'home brew' computers

'71

'72

'73

'74

'75

'76

'77

'78

'79

'80

'81

'82

'83

'84

Motorola

6800

6800: The 8080's rival, with similar capabilities but a completely different design philosophy. Two schools developed: those who preferred the Intel 8080 approach and those who preferred Motorola's 6800 way of working

6809

6809: Motorola's own enhancement of the 6800 is the 6809, arguably the most capable of all eight-bit chips. However, it was too late to have any real impact and as a result has only been used in a few machines

Dragon 32

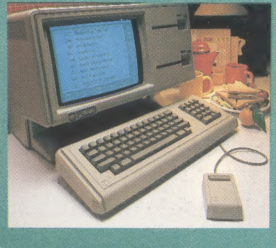
68000

68000: The success of Motorola's highly-acclaimed 16-bit chip has been hampered by the lack of cheap software and the dominance of the 8088. However, Sinclair has chosen the scaled-down 68008 version for its QL

68008

Sinclair QL

Apple Lisa



MOS Technology

6502

Commodore PET

6502: MOS Technology designed its own eight-bit chip, which although not compatible with the 6800, was very much a derivative. Its cheap price made it very attractive to hobbyists and designers and as a result it was used in the first generation of machines such as the top-selling PET and Apple. It remains a popular choice for home micros and has been used in machines such as the Oric and BBC

Apple III

IBM PC

8080

8085

8080: With the arrival of the CP/M operating system, this became the first chip that could be used to build home and business micros. Intel's own update to the 8080 was the 8085, which was 8080-compatible, with extra functions and fewer support chips. The 8085 is available in a low-power CMOS version often found in portables such as the Tandy Model 100

Tandy TRS-80 Model 100

8086

8088

8088 and 8086: The 8088 is a scaled-down version that can use older support chips, and so for a while was the most popular 16-bit chip. Its use in the Sirius and in the IBM PC made it the most popular 16-bit chip. The higher-performance but fully compatible 8086 is now being used in most machines

Zilog

Z80

Z80: A team of designers left Intel to form Zilog and produce this enhanced but fully compatible version of the 8080

Tandy TRS-80 Model 1

Z8000

Z8000: Zilog's first 16-bit chip has not proved popular in the general-purpose computer market. Bad timing and the adoption of the 8088 by IBM may have been the reasons. Zilog's second attempt at a 16-bit chip is the Z800, which promises full Z80 (and hence 8080) compatibility

Olivetti M20

In addition to this innovation, Zilog also came up with another important commercial advantage. Whereas the Intel chip depended on a special clock generator chip as well as a system controller chip, the Zilog team managed to combine all the logic needed for a microprocessor-based computer onto a single chip. Even though it was relatively expensive, the fact that it could replace several other chips made it very attractive to manufacturers.

Although the 6800 had not fared well compared with the 8080, it was still popular among some designers and programmers. Motorola eventually designed a highly sophisticated eight-bit microprocessor called the 6809 that enhanced the 6800. Unfortunately, by the time the 6809 hit the market, a rival company called MOS Technology had come out with a further 6800 enhancement called the 6502. This is the most popular of a number of processors known as the 6500 series. All the members of this series use the same instruction set, but differ in their power and capabilities.

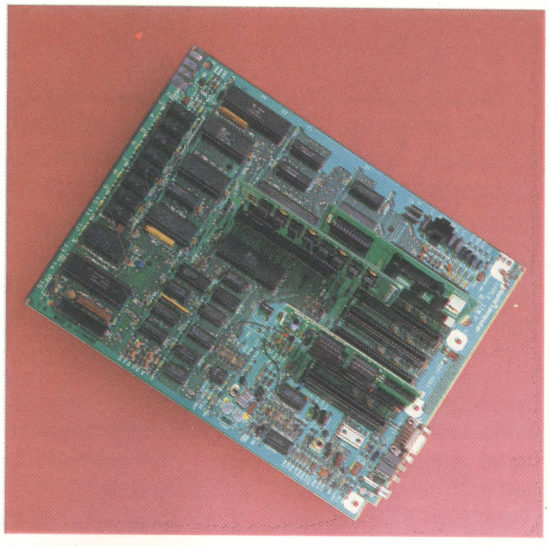
MOS Technology's 6502 follows a design philosophy very close in spirit to Motorola's 6800, but it is not compatible with the 6800 either in terms of hardware requirements, or software compatibility. The Z80, on the other hand, incorporates the entire instruction set of the 8080, and can replace it in a computer system, albeit with some major design surgery.

The 6502 offers an instruction set that any 6800 programmer would feel at home with, advanced capabilities, and slightly easier interfacing requirements. But it provides neither software compatibility, nor the possibility of chip-for-chip replacement. Given these facts, it is hard to imagine that the 6502 would enjoy its present prominent position if it hadn't been for another lucky chance: the 6502 was used in the phenomenally successful Apple computer.

When the Apple appeared, desk-top microcomputers were dominated by S-100 based bus designs. These relied on a 'motherboard' to convey power and signals to a separate board for every function. A minimal S-100 system would therefore require a power supply, a motherboard, a CPU board, a memory board, a VDU board, and probably a printer board and a separate disk drive board. It is therefore easy to see how expensive an S-100 system would be compared with a one-board system such as the Apple.

Relatively cheap though the computer was, the major breakthrough for Steve Wozniak and his team at Apple came with a piece of applications software called VisiCalc. This program proved very popular with businessmen, who found they could use it to generate financial predictions more quickly and easily than with a calculator, pencil and paper. VisiCalc was so successful that it gained Apple massive sales for their computer, and this established the 6502 as one of the leading microprocessor designs. Commodore also opted for the 6502 in the PET and its successors.

Yet a further boost came in the UK when Acorn produced its BBC Micro, also based on this chip. The BBC had originally specified a Z80, but no British manufacturer was able to come up with a suitable design in the time limit set.



Chip Count
Sophisticated chips reduce the number of chips needed on a circuit board. When Apple upgraded the Apple II, the new IIe version had half the number of major chips

While the 6502 chip established its dominance of eight-bit computer design, 16-bit computers began to appear on the market. Intel offered the 8088 and the 8086 for these computers, while Motorola produced the 68000 and Zilog produced the Z8000. All three 16-bit designs have their merits, but none is compatible with their eight-bit predecessors. Fortunately for Intel, Digital Research and Microsoft were quick to come up with operating systems for the 8086/8088 (CP/M-86 and MS-DOS respectively), while Zilog and Motorola were badly served by the software community. IBM's adoption of the 8088 in its PC computer has also given a further boost to the Intel chip.

The fight for market dominance among 16-bit chips promises to be a repeat of the eight-bit chip's history. Intel's 8086 (and the cut-down version, the 8088) have become standards in the same way as the Z80 and the 6502. Chief among the reasons are software support from the MS-DOS and CP/M-86 operating systems, and their selection in top-selling micros, notably the IBM and the Sirius. Zilog's Z8000 chip has only been used in one general-purpose micro – the Olivetti M20. Olivetti struggled to provide the machine with software, finally launching a plug-in card with an 8086 to allow it to run MS-DOS and CP/M-86 software. Since this time, Zilog have set about designing a new chip, the Z800, which is not only 16-bit, but can run software based on the Z80 processor.

In spite of the recent rapid growth in the 16-bit field, the majority of computers currently on sale are based on either the Z80 or the 6502 eight-bit designs. The 16-bit computers undoubtedly offer speed and power advantages over their predecessors, but there's plenty of life left yet in the eight-bit machines, in view of the vast amount of software that has already been developed.

HALF MEASURE

Simple integrated circuits replace numerous transistors in the computer by providing ready-made logic elements in a convenient package. We now progress from the transistor circuits that we used to build AND, OR and NOT gates (see page 144), and use two integrated circuits to build a half adder circuit.

The individual logic gates that we made in the last Workshop project are the basis of more complex digital circuits. One such group of logic gates is the half adder, which we looked at in the Computer Science course (see page 33). This circuit is used to add two single bits. The half adder uses two inputs, the single bits to be added and provides two outputs, the sum and a carry bit. The truth table that represents this is as follows:

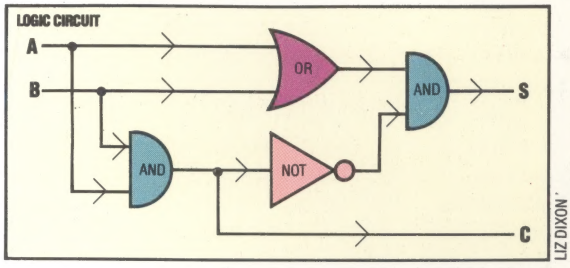
A	B	S	C
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1

The sum output is the sum of the two input bits. When these two bits are both one, the sum is 10 in binary. This result cannot be represented with the single bit output, so the output overflows into the second bit. This overflow is the carry bit.

A half adder is not very useful in eight-bit computers: what is really needed is a circuit to add two eight-bit words together. This circuit can be constructed from 16 half adders. The first two bits are added using the first half adder and its sum bit forms the first bit of the result. Its carry bit is added to the result of the second sum, and the carry from that addition to the third, and so on, thus linking them together.

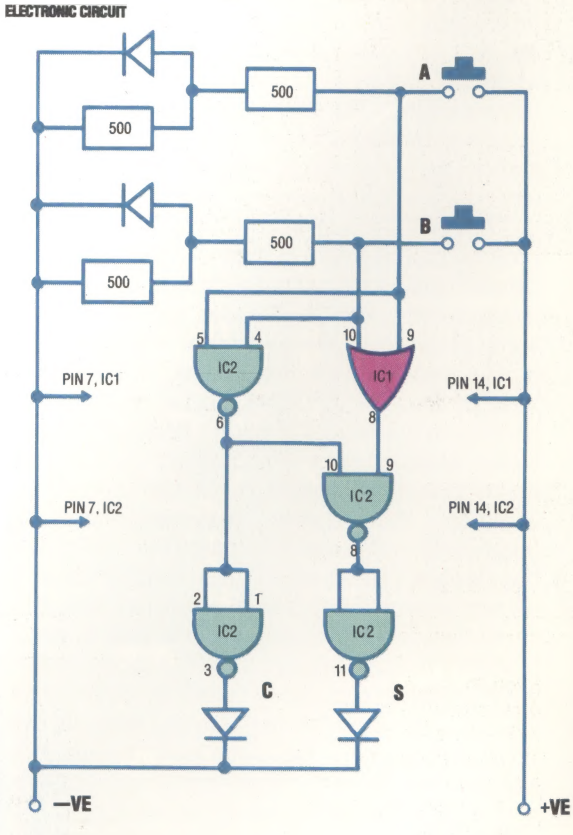
Even a simple half adder would require about 10 transistors in the gates we have already constructed. However, AND, NAND, OR, NOR and other logic gates are available very cheaply in groups of four in single integrated circuits. A half adder can be built more simply from such integrated circuits.

The logic circuit of the half adder is shown opposite. This is the simplest form of the circuit. It uses three kinds of logic gates: OR, AND, and NOT. As the integrated circuits we will be using each contain only a single type of gate, this logic circuit has been simplified to use fewer different gates. The circuit we will build uses four NAND gates and a single OR gate. The number of integrated circuits has been reduced to two. This circuit is more complicated than the single gate



circuit we built on page 144, so special care should be taken to ensure that all the components are placed in the breadboard correctly.

Once you have built this circuit, you may consider it to have been a lot of hard work to achieve very simple results. Although it is much easier than building the circuit from discrete



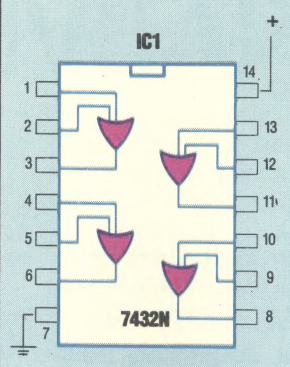
components such as transistors, it is hard to conceive of an entire computer built in this way.

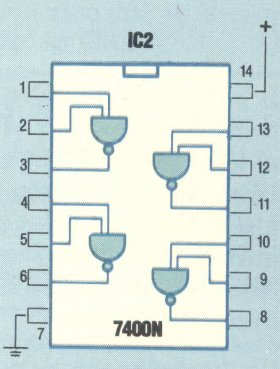
In practice, chips are rarely ever used in this way and only occasionally crop up doing a menial task in the corner of a circuit board. Larger chips have more signals going into and out of them so that the whole chip is a complete device that will, for instance, add together two four-bit numbers.

The level of complexity grows until particular chips are capable of performing whole tasks by themselves.

What You Need

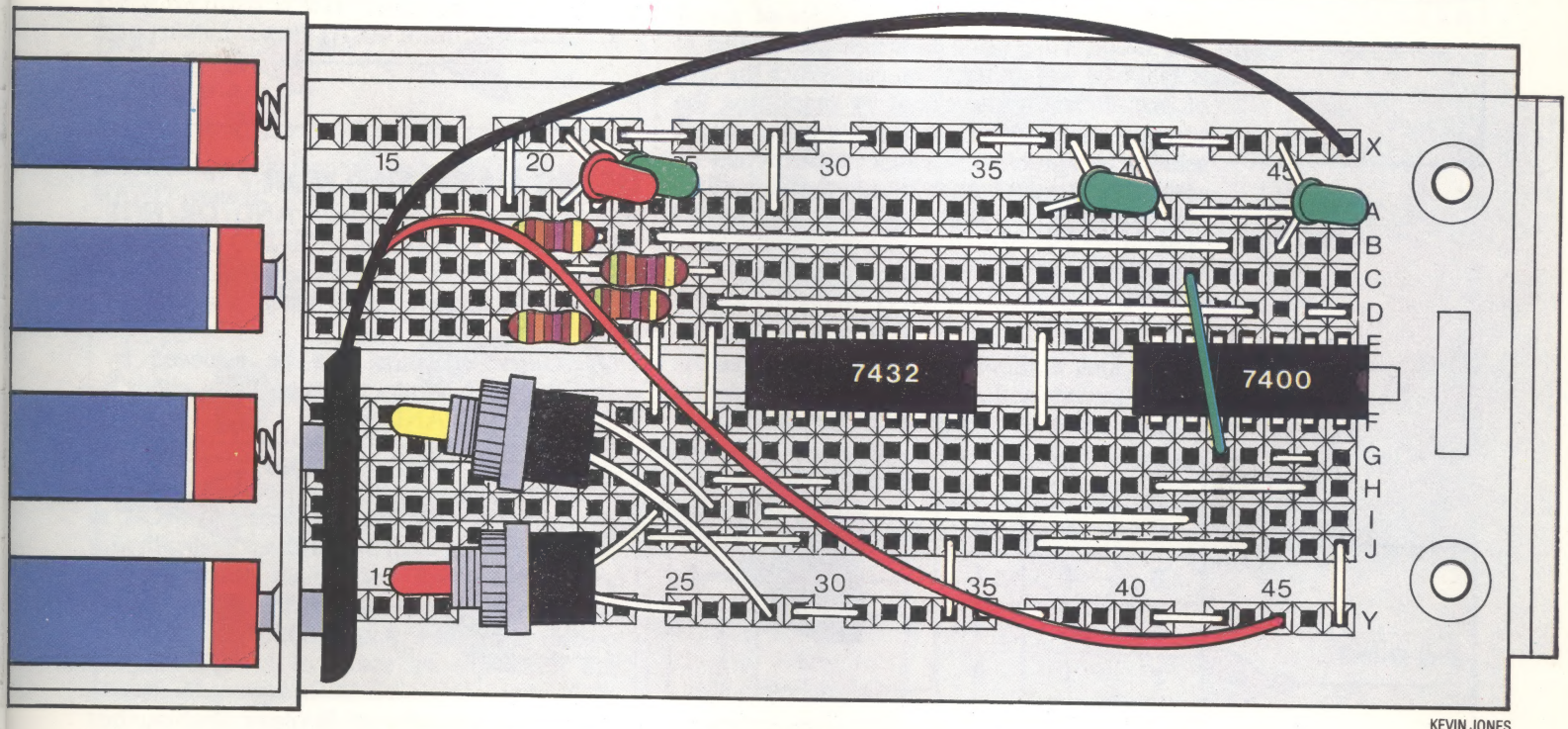
4 500-ohm, one-quarter watt resistors
4 LEDs
2 push-to-make switches
1 7400N integrated circuit
1 7432N integrated circuit
4 HP7 or equivalent batteries
1 battery holder
1 battery clip
1 breadboard (Experimentor 300 or similar)
Short lengths of wire





Pinouts

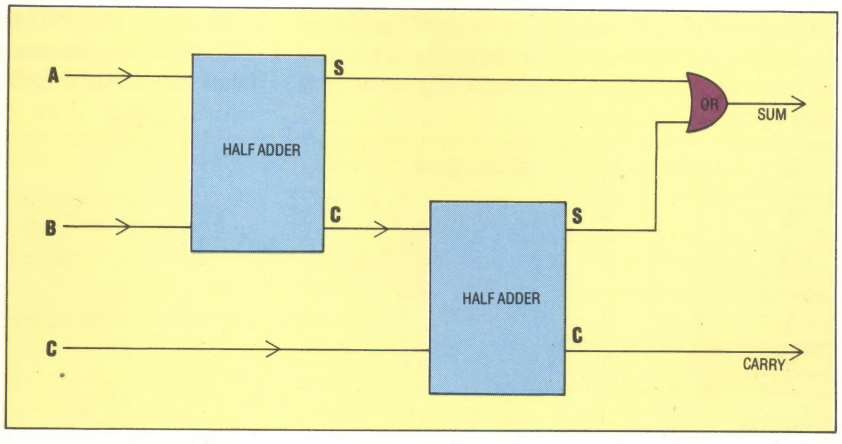
The two integrated circuits we will use are 14-pin TTL chips. The 7400N is four NAND gates on a chip and the 7432N provides four OR gates. This diagram shows the way the gates are arranged, all that is needed is to connect the appropriate pins together. There is a notch on the top of the chip to show which way up to use it. Remember also that you must supply power through pins 7 and 14 to make the chips work



KEVIN JONES

Across The Board

Once the electronic circuit is designed, the next step is the process of arranging the components on a breadboard. You can buy planning pads for this or simply use a photocopy of an empty board! It is best to keep the breadboard looking as much like the original circuit as possible, as the neater the design, the easier it is to build. Copy this precisely, as all the components are in the correct position



Full Adder

As an exercise, you might like to try extending your half adder circuit to a full adder. This circuit not only adds two bits together, but also adds a carry from any previous bit position. A series of full adders can add complete binary words. The simplest way to create a full adder is to build two half adders as shown here. The sum signal from the first half adder must replace one of the input switches on the second half adder. The carry output from the first adder must be ORed with the carry output from the second adder to produce the signal for the carry LED

ALTERNATIVE PATHS

It is possible to solve any logical problem using combinations of the three basic types of logic gates (AND, OR and NOT) that we have met so far in the course. In this instalment of the Logic course, we introduce two new gates — NAND and NOR — which give us alternative ways of designing circuits.

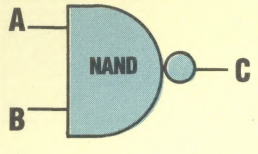
If we can solve all logic problems using AND, OR and NOT gates, why do we need to bother ourselves learning about other types of gates? The reason is that using these new gates, either in isolation or with other gates, can reduce the cost of manufacturing the circuit by simplifying the wiring required or by producing a more elegant solution to a problem. All logic problems may be solved using one of the following techniques:

a) AND, OR and NOT gates together
b) NAND gates only
c) NOR gates only
d) a combination of the above

So let's look at these two new types of gates. As with all circuits and circuit elements, the function of each gate is best described by its truth table.

A	B	C
0	0	1
0	1	1
1	0	1
1	1	0

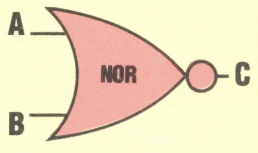
THE NAND GATE



NAND is short for Not AND, and comparing this truth table with the one for an AND gate (see page 8) it can be seen that in the output column all the ones have been exchanged for zeros, and vice versa.

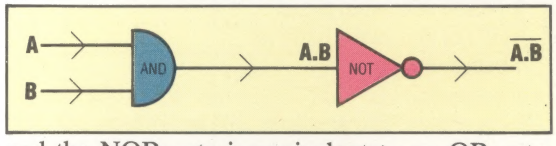
A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

THE NOR GATE

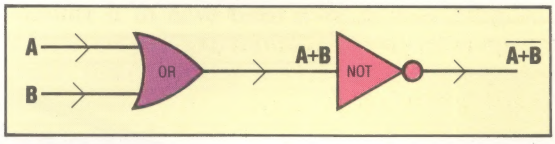


Similarly, NOR is short for Not OR, and a comparison of the output columns for this table and the table for an OR gate (see page 8) again shows that all the ones and zeros have been negated.

There are no special symbols for NAND and NOR operations in Boolean algebra but we can represent each function using the AND, OR and NOT symbols that we have already met. A NAND gate is equivalent to this simple circuit:



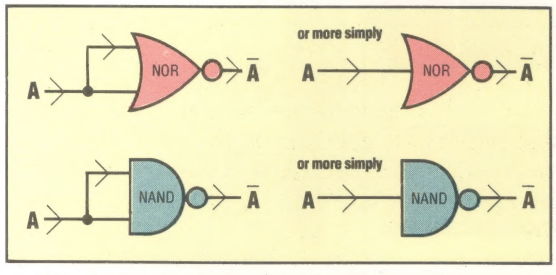
and the NOR gate is equivalent to an OR gate followed by a NOT gate:



USING NAND AND NOR

Just as it is possible to draw AND/OR/NOT circuits that are equivalent to NAND and NOR, so we can represent each of these three basic gates in terms of a series of NOR gates or a series of NAND gates.

NOT Gates: Negation can be achieved by connecting both inputs together, using either a NOR gate or a NAND gate:

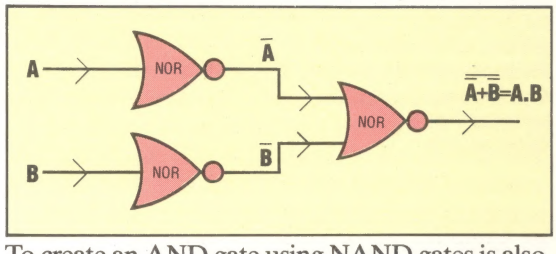


AND Gates: In terms of Boolean algebra, the output from an AND gate with inputs A and B is A.B. However, we can manipulate this expression into a more useful form:

$A.B = \overline{\overline{A}.\overline{B}}$ (as $A = \overline{\overline{A}}$)

$= \overline{\overline{A} + \overline{B}}$ (de Morgan's Law)

Thus the circuit can be made by putting NOT(A) and NOT(B) through a NOR gate:

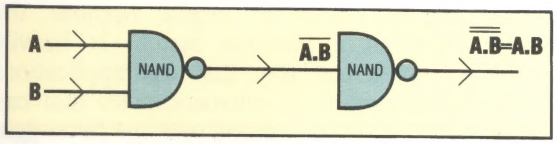


To create an AND gate using NAND gates is also possible. The output from a NAND gate is $\overline{A.B}$.

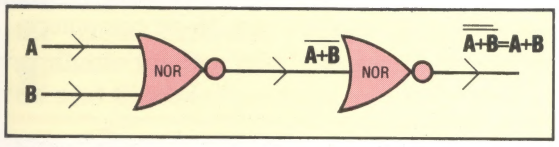
If this output is negated then we will get:

$$\overline{\overline{A.B}} = A.B$$

So the circuit will be:



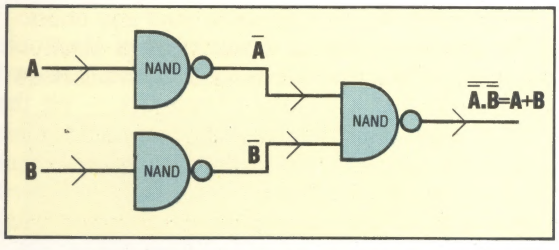
OR Gates: Just as chaining two NAND gates together is equivalent to an AND gate, so if we chain two NOR gates together we obtain a circuit that is equivalent to an OR gate:



The required output from an OR gate is A+B. Using the rules of Boolean algebra, this can be manipulated into a NAND form:

$$A+B = \overline{\overline{A}+\overline{B}}$$
$$= \overline{\overline{A}.\overline{B}}$$

and consequently the corresponding circuit using NAND gates is:

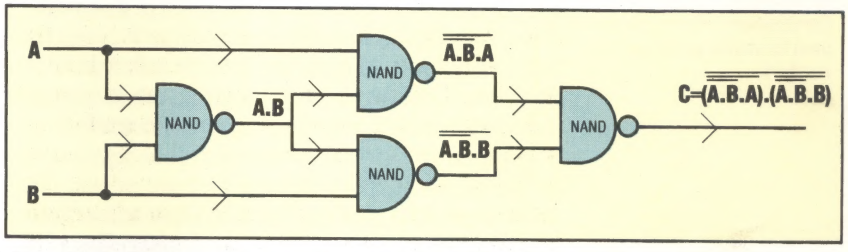


If we wish to construct a circuit using only NAND or NOR elements then we may still follow the simplification methods we have already met, but first we must manipulate the final Boolean expression into a form that is suitable. For circuits incorporating NAND gates, we use the rules of Boolean algebra to create an expression that consists of groups of ANDs connected by ORs, and use de Morgan's theorem repeatedly until the expression is completely in NAND form. For circuits in the NOR form, we employ similar rules, as the example will show. To demonstrate how these rules are used let's look again at the Exclusive OR (XOR) gate (see page 47). The output from an XOR gate can be defined by the expression $C=\overline{A.B}.(A+B)$.

Let us take this expression and convert it so that a circuit for the XOR gate may be constructed solely from NAND gates. First of all, let's manipulate the expression so that we obtain groups of ANDs connected by ORs.

$$C = \overline{A.B}.(A+B)$$
$$= (\overline{A.B}.A)+(\overline{A.B}.B) \text{ (multiply out brackets)}$$
$$= \overline{\overline{(A.B.A)}.\overline{(A.B.B)}} \text{ (de Morgan's theorem)}$$

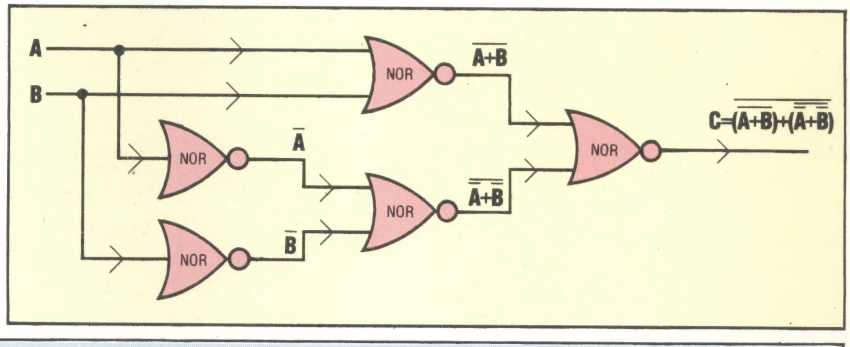
When drawing the circuit from a complicated expression such as the one above, it is best to start from the output and work backwards to the inputs. Try following this circuit diagram from output to input to see how it was constructed.



For the NOR form, we must again start with the original simplified expression for the XOR gate and manipulate it into groups of ORs connected by ANDs. This first step can be done by using de Morgan's theorem on the left hand part:

$$C = \overline{A.B}.(A+B)$$
$$= (\overline{A}+\overline{B}).(A+B)$$
$$= \overline{\overline{(\overline{A}+\overline{B})}+(\overline{A}+\overline{B})}$$

Converting this expression into a circuit diagram is again best done by starting at the output and working backwards.



Answers To Exercise 6 On Page 147

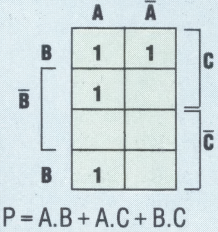
1a)

Inputs			Output
A	B	C	P
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

b) The Boolean expression for P is:

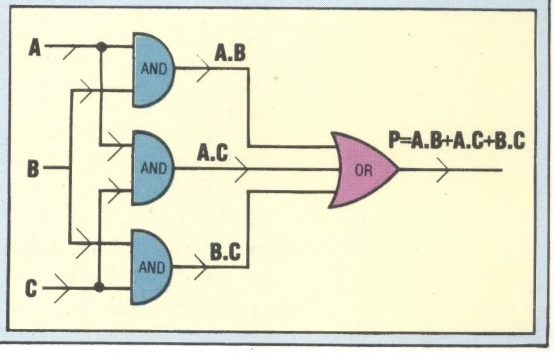
$$P = \overline{A}.B.C + A.\overline{B}.C + A.B.\overline{C} + A.B.C$$

Simplification may be achieved by using a Karnaugh map:



Thus $P = A.B + A.C + B.C$

c) The circuit is:



LIZ DIXON

B

BASIC

All home computer owners should be familiar with this term, and most will know that it supposedly stands for Beginners All-purpose Symbolic Instruction Code. Its origins are less well known. BASIC was developed at Dartmouth College, USA, not as a language for developing software but for teaching programming.

It was really a derivation from FORTRAN, the most popular language among scientists, engineers and academics at the time. BASIC cut down on the complex syntax of FORTRAN (and on its range of functions). In particular, it replaced the hard-to-use WRITE and FORMAT statements with the simple PRINT command.

But the major breakthrough was that BASIC was designed to be interactive: typed in and operated by someone at a terminal, rather than as a stack of pre-punched cards. This is why all BASIC lines have line numbers, so that they can be referred to and edited. Editing a FORTRAN program had meant finding and altering the appropriate cards.

Flip-Flop
A bistable or flip-flop can store or 'remember' a single bit. Combinations of these can be used for storing whole binary numbers

BCD

Binary-coded decimal (or *BCD*) is a method of storing decimal numbers in binary form (for example, in RAM or on disk). Most home computers, however, favour the floating point format in preference to BCD, because it is more efficient in terms of memory usage. Using floating point, a number is converted into one long binary number and then normalised (the radix point — see page 148 — is shifted and the number separated into a mantissa and an exponent). It is then stored in a pre-determined number of bytes, which on home computers is commonly five.

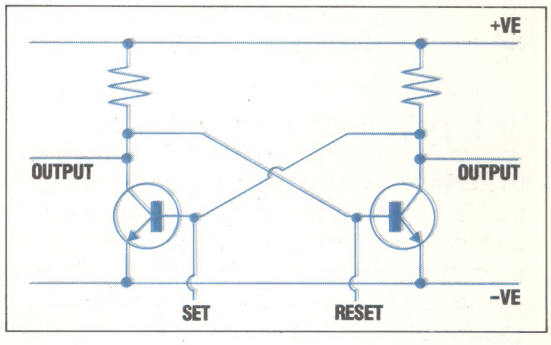
With BCD, each digit in the original decimal number is converted into a four-bit binary number (half a byte), so the number of bytes occupied will correspond to half the number of decimal digits. The computer performs all arithmetic on BCD numbers in a way that is very similar to the way that we perform long multiplication or division (working on each digit of the number separately), whereas a floating point subroutine would treat the number as a whole.

The major advantage of BCD is that it doesn't produce the kind of rounding off errors that we often associate with computers and pocket calculators. This can be particularly important in major banking and financial applications.

BENCHMARK

In the early days of microcomputing (when the PET, Apple II and Tandy TRS-80 were the predominant machines) a set of *benchmarks* was developed to determine the relative speed and efficiency of the BASIC interpreters. These consisted of 10 simple routines that tested different aspects of the BASIC (the speed of execution of loops, floating point arithmetic, trigonometric functions, etc.). You can still find the results of these tests printed in magazines that undertake technical reviews, where they take the form: 'BM1 – 10.2 seconds, BM2 – 3.87 seconds...' and so on.

Attempts to introduce a parallel system for modern business microcomputers have met with little success. This is largely because the throughput of a business system is heavily dependent on the way that the applications software is written. A microcomputer that can execute the XYZ accounting package faster than any other machine may well be the slowest on the ABC database. The eight-bit Osborne 1, for example, is not renowned as being a fast machine, yet it is liked by many journalists because it can execute the Wordstar word processing program faster than most of the new 16-bit computers.



LIZ DIXON

BISTABLE

The *bistable* is one of the simplest of electronic circuits — you can construct one from just two transistors and a handful of resistors — yet the microcomputer owes its very existence to this invention. As the name suggests, a bistable circuit is one that has two stable states, usually indicated by an output line that is 'high' (around five volts) or 'low' (zero volts).

The bistable circuit can therefore be thought of as a single memory bit, capable of storing a '1' or a '0'. The first solid-state semiconductor memories consisted of banks of transistors configured to form an array of bistables. Static RAM chips, which are still found in quite a few microcomputers, are nothing more than miniaturised arrays of bistables. However, modern designs tend to favour dynamic RAMs, which store the information in the form of electrical charges applied to tiny capacitors. As these charges tend to leak away, they have to be constantly refreshed by a special electronic circuit built into the chips. However, dynamic RAMs are faster, and consume less current, than their predecessors.

Bistables can still be found in the discrete logic section of your computer's printed circuit board. They are colloquially known as *flip-flops*, because of their ability to alternate between two states. Flip-flops differ in the way that the state is changed: some have one input line and change state whenever a pulse is applied. But the most common form is the *J-K flip-flop*, which has two inputs (labelled J and K). Applying a voltage to J will initiate one state, while a voltage applied to K will give the other.

SCRIBBLE PAD

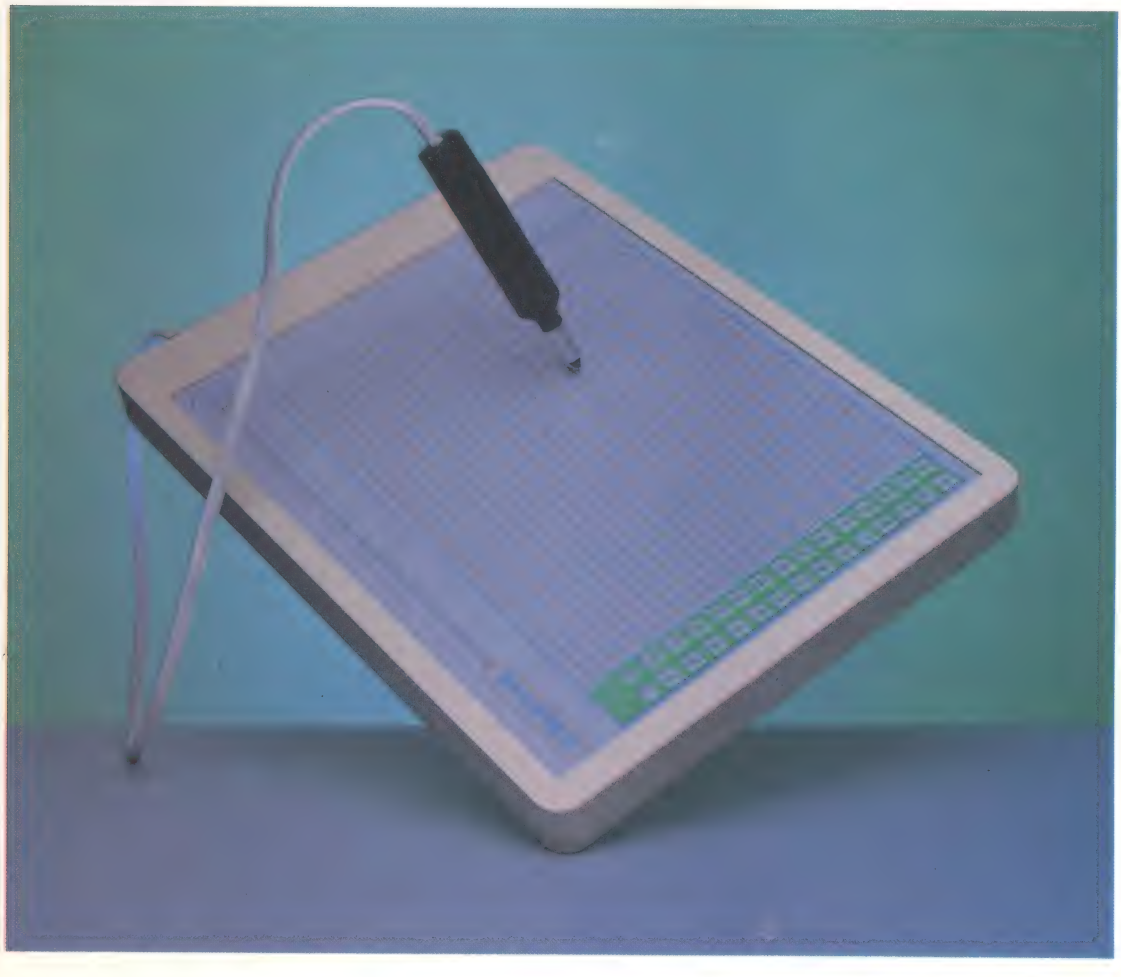
The Grafpad is a digitising tablet for producing detailed designs and drawings on a home micro. It offers a basic specification at a price low enough to attract people who would normally avoid this area because of the costs involved. There are versions available for the BBC Micro, Commodore 64 and Sinclair Spectrum.

Graphics tablets are one of the most versatile and useful peripherals for micros. They have obvious uses as drawing and design aids, from freehand art to electronic circuit design and tracing maps. But beside straightforward drawing applications, they provide a useful extra input device. A card overlay on the graphics tablet can have all of a program's features laid out, either in words or pictorially. All you do is touch the appropriate command with the stylus (pen), and the software will work out which option you have selected.

Such systems used to be the preserve of specialist machines, sold specifically for designers and engineers. But prices have fallen sufficiently to let home users try tablets out for themselves. The Grafpad examined here is one of the leading low-cost designs, bringing a good specification for a reasonable price. It's available in specific versions for the BBC Micro, Commodore 64 and Sinclair Spectrum. The version illustrated here is for the BBC.

There are three elements to the Grafpad: the pad itself, a linked stylus and the controlling software. The pad connects to the BBC via the user port and the stylus plugs in to its side. The surface of the pad is divided into a ruled grid of 16 by 20 boxes and a command bar (a separate panel with single letters inscribed on it). The command bar can be used to control some of the software without the need to use a keyboard. On top of this slots a perspex cover to protect the surface of the pad. It is possible to design your own 'overlays' with your own commands and grids drawn on them.

Inside the pad is a grid of 320 by 256 wires approximately 1.2mm apart. The stylus nib is a

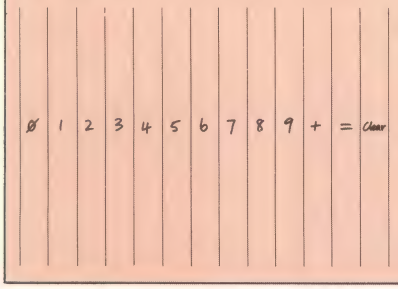


Graphic Ideas
The Grafpad can be used with its own software to create designs and drawings, or with your own programs as an input device

Adding Machine

We wrote a simple program on the BBC Micro to use the Grafpad as an input device for an adding machine. An overlay with all the number keys for the adding machine is placed under the pad's perspex cover. Touching the relevant key with the stylus will operate the software

```
 10 REM GRAFPAD DEMO
 20 REM
 30 REM An adding machine using the Grafpad
 40 REM
 50 MODE 1
 60 PRINT "ADDING MACHINE":PRINT
 70 HIMEM=&29FF
 80 XADJUST=7
 90 REM
100 REM set up co-ordinate table
110 REM
120 DIM B(15):FOR I=1 TO 15:B(I)=I*25:NEXT I
130 R1=0:R=0
140 REM
150 REM load Grafpad driver program
160 REM
170 *LOAD "PADREAD" 2A00
180 PEN%=&2A00
190 REM
200 REM MAIN PROGRAM STARTS HERE
210 REM
220 REM wait for pen to be pressed
230 CALL PEN%
240 !X%=!X%-XADJUST:IF !X%<0 THEN !X%=0
250 IF ?U%>0 THEN 230
260 REM beep to register pen press
270 SOUND 1,-15,120,1
280 X=!X%
290 REM convert position to 0-12
300 I=0
310 IF X>B(I) THEN I=I+1:GOTO 310
320 REM interpret codes 0-12
330 IF I<10 THEN R=R*10+I
340 IF I=10 THEN PRINT R;" +":R1=R1+R:R=0
350 IF I=11 THEN PRINT R;" =":PRINT "      ____":R1=
R1+R:PRINT R1:R1=0:R=0:PRINT:PRINT
360 IF I=12 THEN PRINT:PRINT"CLEAR":PRINT:R=0:R1=0
370 REM wait until pen is lifted again
380 CALL PEN%
390 IF ?U%=0 THEN 380
400 REM loop for next pen press
410 GOTO 230
```

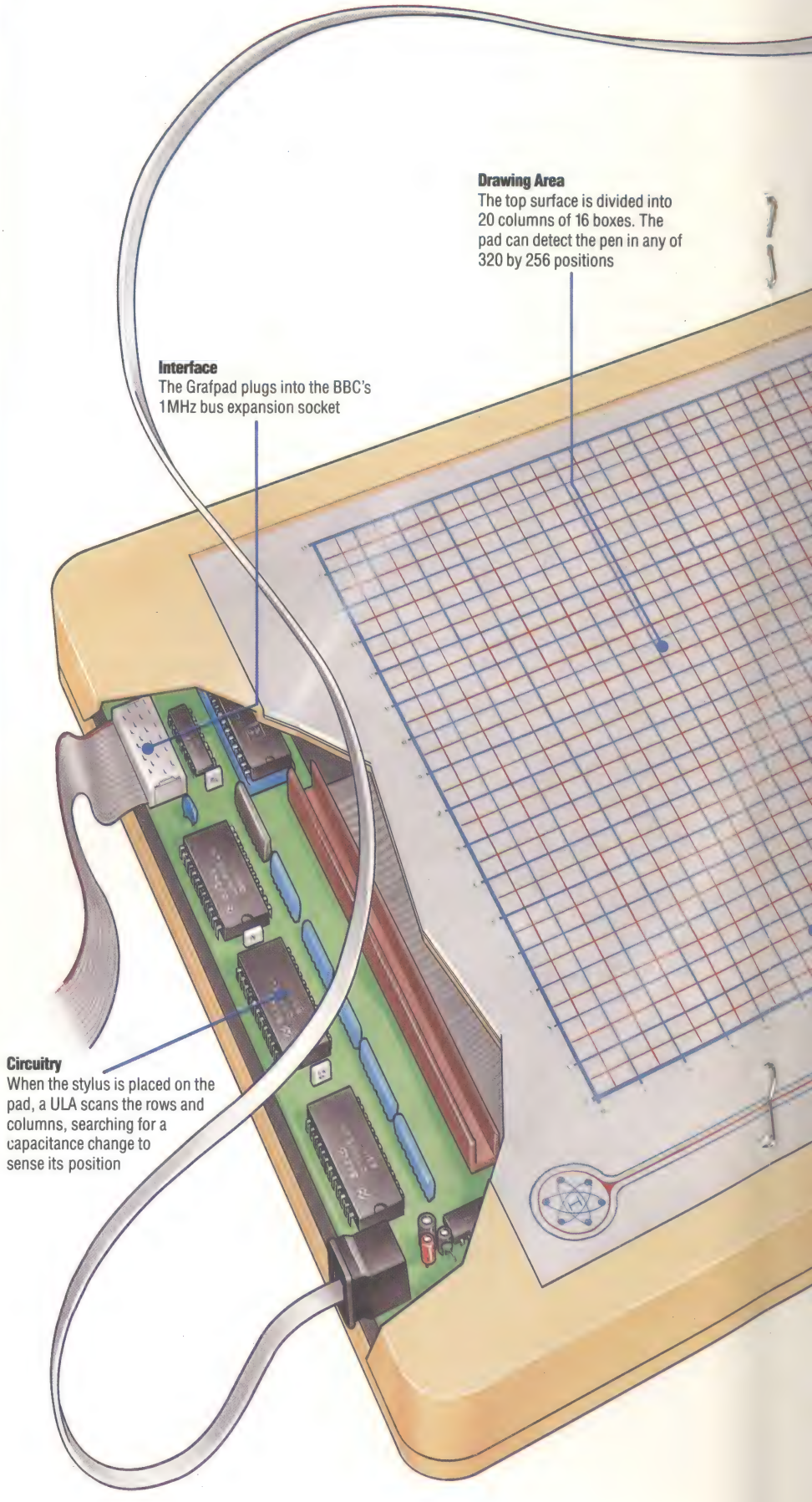


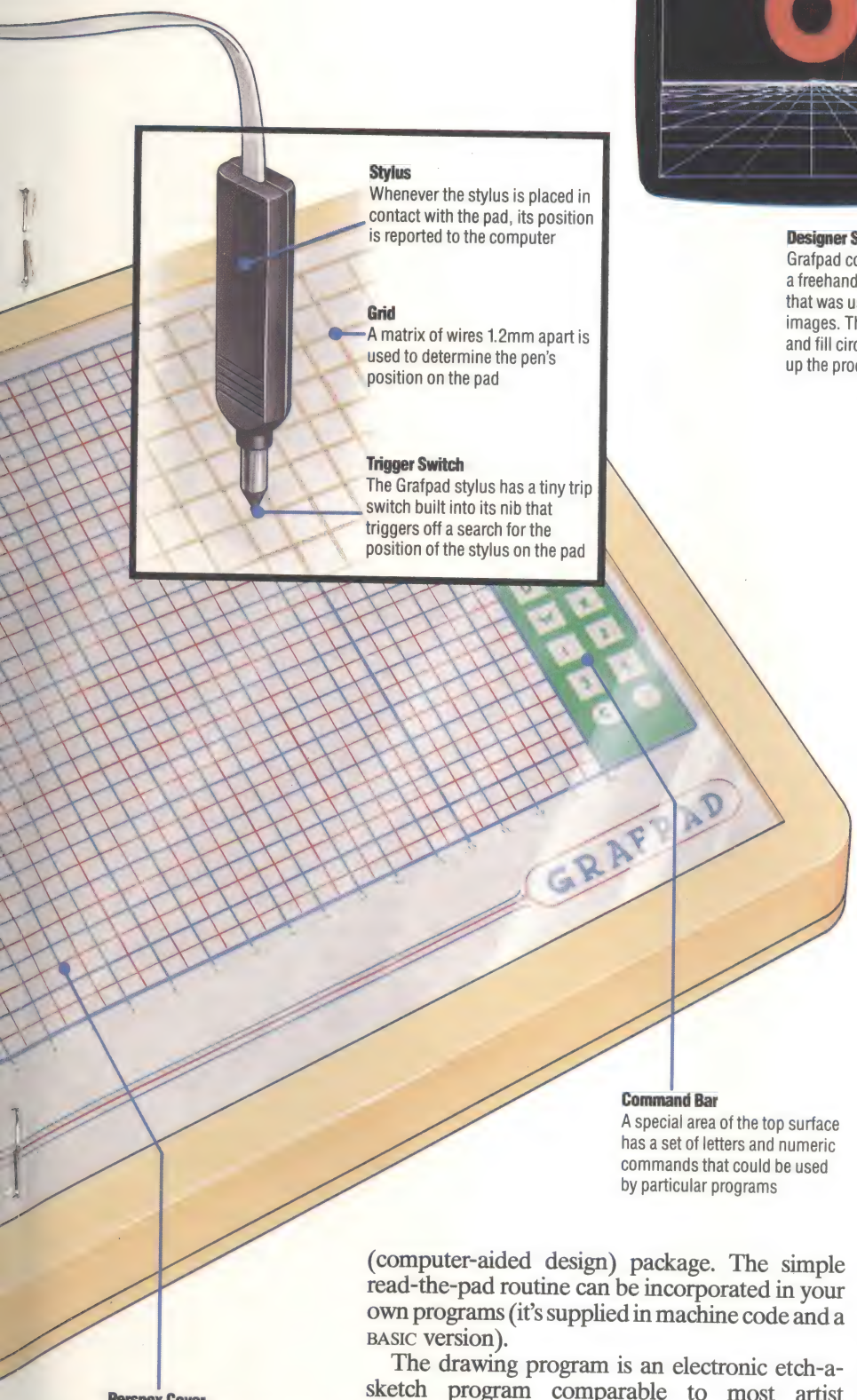

```
ADDING MACHINE

 34 +      4 +
 96 =      8 =
____     ____
130       12

  3 +  19876 +
  9 +   8876 +
  6 +   8864 +
 28 +   1243 =
126 =
____   _____
172    38859
```

tiny switch. When you push the stylus down on the perspex cover of the pad, a ULA (uncommitted logic array) chip pulses each of the wires in turn until it detects the position of the pen by a change in capacitance. This scanning takes place 2,000 times a second, making locating the stylus a very fast process. The stylus should be held by the earthed metal band around its nib to help the system work reliably.

When the stylus is placed on the pad, the computer receives the 'stylus down' signal and a report of its co-ordinates on the pad. The exact effect it creates is determined by the software. A cross-shaped cursor might appear on the screen in a corresponding position, or a particular command might be triggered. It's here that the Grafpad's economy begins to show. The stylus can only be detected on a grid of 320 by 256 positions, making it difficult to draw very smooth or fine detail. The pad is also quite small – a sheet of A4-sized paper is a sensible work area.

The Grafpad has three software packages; ranging from a simple demonstration routine, via a simple drawing program, to a complex CAD


Drawing Area
The top surface is divided into 20 columns of 16 boxes. The pad can detect the pen in any of 320 by 256 positions
Interface
The Grafpad plugs into the BBC's 1MHz bus expansion socket
Circuitry
When the stylus is placed on the pad, a ULA scans the rows and columns, searching for a capacitance change to sense its position



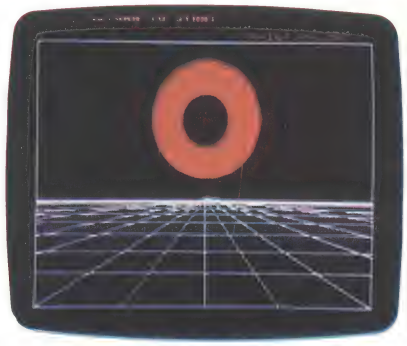
Stylus
Whenever the stylus is placed in contact with the pad, its position is reported to the computer

Grid
A matrix of wires 1.2mm apart is used to determine the pen's position on the pad

Trigger Switch
The Grafpad stylus has a tiny trip switch built into its nib that triggers off a search for the position of the stylus on the pad

Command Bar
A special area of the top surface has a set of letters and numeric commands that could be used by particular programs

Perspex Cover
A perspex sheet protects the top of the pad. Overlay sheets can be taped to this



Designer Software
Grafpad comes with PROG2, a freehand drawing package that was used to create these images. The ability to draw and fill circles greatly speeds up the process

(computer-aided design) package. The simple read-the-pad routine can be incorporated in your own programs (it's supplied in machine code and a BASIC version).

The drawing program is an electronic etch-a-sketch program comparable to most artist packages available, even those that don't use a pad. It offers all the basic features: lines, boxes, circles, triangles and 'freehand', and will fill an enclosed area with a particular colour. However, it lacks more sophisticated facilities, such as being able to copy and move sections of the drawing. Certainly, there is nothing here that a keyboard-only piece of software couldn't do although the Grafpad does allow designs to be traced. The BBC version will display only four colours at once and suffers from slow response times.

The CAD program is simply a demonstration of some of the principles involved. First, you create a set of characters to be used in the construction of your designs. For electronics, these shapes might be components such as transistors, resistors and so on. You could also create logic gates, pieces of furniture, or even tile patterns. Once these are created, you move on to the actual drawing board where you can freely repeat and arrange the shapes and join them with straight lines.

This is very similar to how a real CAD package works. But the Grafpad software isn't up to serious use. Among the facilities you would need are the ability to label the diagrams, rotate and scale drawings, magnify a particular portion of the screen, position small shapes very accurately and so on. More flexibility in correcting mistakes is essential and in general the CAD program misses the point of using the Grafpad as an input device. Despite the small command bar on the tablet, many commands need keyboard input and overall operation is rather cumbersome.

The Grafpad itself is a versatile peripheral that offers very good value for money. In terms of area, resolution and reliability, it is restricted in order to be economically priced. However, the software that comes with the system is disappointing and the unit will appeal most to those who want to write their own programs. Even so, with suitable effort, tablets like these will enable people to explore new possibilities and should prove a considerable boost for more advanced graphics on home micros.

TAKING STOCK

The first three articles in this series have looked at how the cash flow of a small business can be controlled by the computer. Now we turn to how the supply and demand of goods can be efficiently monitored. We have chosen Dragon Data's Stock Recording System and two programs for the Sirius as examples of stock handling packages.

In a perfectly run business, where the owner or manager knows exactly what customer demand will be, and what is currently in stock, over- or understocking would never occur. They are both the results of poor information. Computerised stock systems are an excellent way of avoiding poor information.

To carry out the task of stock control, computers have to provide a variety of answers for management. The business needs to know what stock it has, how fast (or slowly) particular lines are moving, when it will need to reorder, as well as the value of what is currently in stock.

The computerised stock system aims to monitor stock movements. These movements can be broken down into the following categories: outgoing stock that is issued to meet sales orders; incoming stock that is bought in from suppliers; stock allocated to meet orders; and stock on order.

To these four categories has to be added the ability to make adjustments to stock levels for goods returned by customers, or for goods sent back by the business to its suppliers — in other words, reject goods. Stock-taking also frequently turns up discrepancies between what is actually on hand, and what is supposed to be on the shelves.

In addition, the system has to keep track of stock values. So as well as recording quantities and monitoring stock movements, the program has to handle price information.

Stock control systems fall into two rather different types, depending on whether they are intended for small businesses in the retailing or distribution fields, or for manufacturing companies. In the latter case, the stock system usually has to take into account the fact that various components will be drawn from stock during the manufacturing process and will be assembled into one manufactured unit. Many microcomputer-based stock control systems try to cover the needs of both types of business. In this article, we will concentrate on the retailing and distribution type of business.

COURTESY OF J. SAINSBURY PLC.

Keeping The Shelves Filled
Automated tills can read product information directly off bar code labels and record the sales on a central stock control computer. Such instant feedback allows big stores to ensure that the shelves and warehouses contain the right products in the right quantities

Since stock control is bound up with so many aspects of a business's activities, it is usual for stock systems to *integrate* with a number of other programs ('integration' means that two or more applications packages will allow values and data to be passed from one application to another). A typical, fully integrated system might be linked to the purchase ledger, the purchase order processing system, an invoicing module, the sales ledger and sales order processing.

Integration has a number of advantages. Take a business, for example, which has a stock control system integrated with its sales order processing system. If these two systems are able to communicate with each other, the stock files can be automatically updated at the same time as the sales order is created. Then too, if the sales order system can look up the stock file for a full narrative description of a stock item and its selling price as soon as the stock code is entered, the operator will have less data to input — and less opportunity for making erroneous entries.

The starting point for any stock control system is, of course, the stock data file. Every system will have a way of identifying all stock lines by a unique code number and by a narrative description. The code number is used by the program as a filing key.

This can lead to a relatively simple stock system, or, in the more sophisticated packages, it can be rather complicated. Dragon Data's Stock Recording System, for the Dragon 64 with a floppy disk drive, provides an example of the simpler sort of system.

This package allows the user eight alphanumeric characters for the stock item code, plus a two-digit product group code. This means that any stock item can be assigned to one of 50 product groups (the maximum that the system will cater for). If any stock item is given an item code of less than eight digits, say 445, the system automatically right-justifies the number. This means that entering 445 is the same as entering 00445 or 00000445.

The point about justification is important. ACT

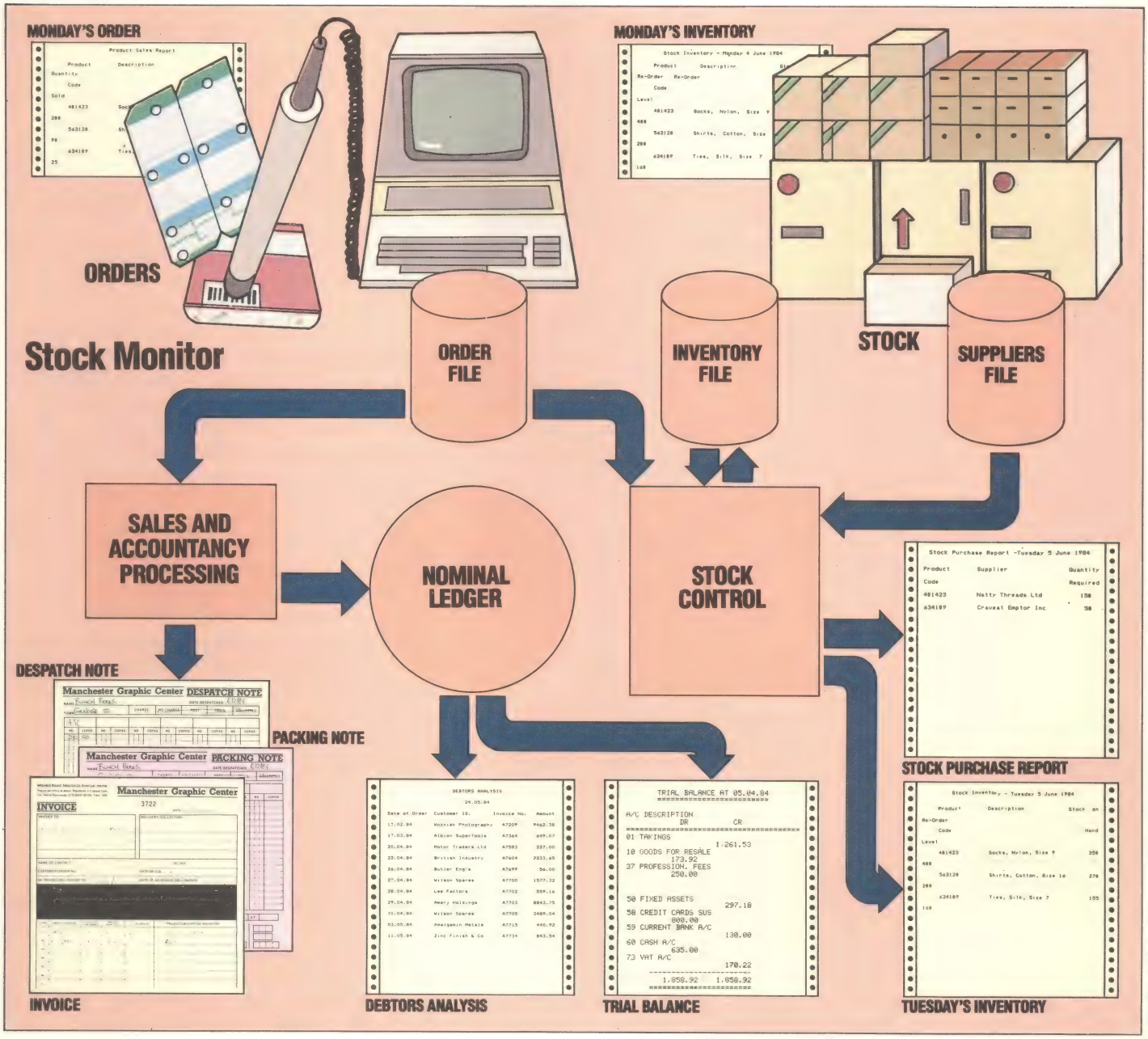
Pulsar's Stock Control System, for example, which runs on larger micros like the IBM PC and the Sirius, offers users the choice between a right- and a left-justified coding system. The resulting stock coding systems are totally different and incompatible.

The product code can be up to 16 alphanumeric characters long. The right-justified system is a numerically ordered system of codes. The left-justified system is designed for users who have rather more complex coding systems, involving alphanumerics — say, PX445/44. It allows codes of different lengths for different products and is useful for systems where the user wants to use the product code to identify some feature of the stock line, such as pattern, size or colour.

Omicron's Powerstock package, which runs on the Sirius, is a more expensive system designed for users with more complex requirements. This has an even more complex coding system, and is defined in terms of stock groups. A stock group can be any set of stock records that are related by common processing or reporting requirements. What makes it different from Pulsar's left-justified coding system is that each stock group is processed separately and different processing rules can be assigned to each group. Remember that whatever the code number assigned to a product line in Pulsar, all code numbers are processed identically.

The coding structure of stock control systems, therefore, has to be flexible enough to allow users to identify and subdivide their stock lines. The simpler systems running on the cheaper home computers tend to offer less flexibility because of the constraints, once again, on memory and storage. The Dragon Data system, for example, is designed to handle a maximum of 350 stock items. Omicron's Powerstock system is open-ended — the maximum number of items depends on the user's computer configuration.

Stock Control
Integrated sales and stock control systems in high turnover businesses can benefit greatly from an automated point-of-sale system to keep the inventory and accounting files up to date. Sales data can be written on a marker such as a Kimball tag or a bar code attached to the product. The markers are read by an optical reader attached to the cash till. This may be a microcomputer itself, or it may pass the data to a computer for processing



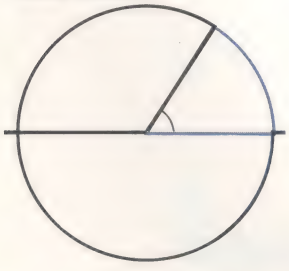
TONY DUNCAN-SMITH LIZ DIXON

DEGREES OF PRECISION

In this second instalment of a series on mathematics and BASIC programming, we continue our look at trigonometrical functions (begun on page 154). Here we look at how the sine and cosine functions can be used in BASIC programs, and also provide ways of testing these two functions to check for any possible sources of error.

Because BASIC is provided with both COS and SIN functions, calculating the position of a point on a line after rotating it through a certain number of degrees should be an easy task. The COS of θ will give the position on the x-axis (the x co-ordinate) and the SIN of θ will give the position on the y-axis (the y co-ordinate). However, when using these two functions, it is important to remember that most versions of BASIC work in radians and not degrees. Another thing that should be checked is that the values returned for θ may not be reliable as θ approaches 0 or 1. The first thing we will do is deal with the vital difference between degrees and radians.

Radian Measure



If a portion of a circle (called an *arc*) is drawn so that its length is exactly equal to the radius of the circle, the angle at the centre is defined as one radian (see the illustration). If the radius of the circle is one unit, this portion of the circumference will also have a length of one unit. The formula for finding the circumference of a circle is $2\pi r$, so there must be 2π radians in one complete revolution. One complete revolution — the turn needed to make a full circle — expressed in a more familiar notation is 360 degrees. Therefore, 360° is equal to 2π radians. This gives us an easy way of relating degrees to radians:

$360^\circ = 2\pi$ radians
$180^\circ = \pi$ radians
$90^\circ = \pi/2$ radians
$1^\circ = \pi/180 = 0.0174$ radians

A BASIC program that needed to find the cosine of an angle measured in degrees would first have to convert the angle measure from degrees into radians, and then use the COS function. Try this:

```
10 INPUT "INPUT ANGLE IN DEGREES";A
20 LET B# = A * 0.0174
30 LET C# = COS(B#)
40 PRINT "THE COSINE OF ";A;" DEGREES IS ";C#
50 END
```

The hash symbols indicate that the variables in the program are double precision (which we'll look at later in this article). A simple modification of this program using the sine function, will input all values of θ from 0° to 360° and produce the sine of these values as a table. If these values are plotted against the y-axis of a graph (where the x-axis represents values of θ in radians), the sine wave graph familiar to hi-fi buffs and electrical engineers will result (see the diagram on page 155). This familiar curve is nothing more than the plot of positions of the intersection of the hypotenuse with the unit circle on the y-axis for all angles of rotation. In other words, it is an alternative way of describing a circle mathematically.

A few versions of BASIC allow the SIN and COS functions to work on either degrees or radians by using a 'software switch', but most do not. If you prefer to work in degrees all the time, it is possible to define a 'user defined function' to make the conversions for you. Here is one possibility:

```
10 REM A USER DEFINED FUNCTION FOR WORKING
   IN DEGREES
20 DEF FNDSIN (D#) = SIN(D#*0.017453293)
30 INPUT "INPUT ANGLE IN DEGREES";D#
40 PRINT "THE SINE OF";D#," DEGREES IS";
   FNDSIN(D#)
50 END
```

Line 20 defines a function called DSIN (standing for 'degrees/sine') that uses as its only parameter the double precision variable D#. The right hand half of the definition simply shows how the value to be returned by the function (the sine of an angle in degrees) is to be derived. To call a user defined function, you simply use the name of the function (with the value to be operated on in parenthesis) as usual. Note, however, that the line containing the definition must be executed before any calls to the function can be made.

One of the problems of using the sine function in BASIC is that not all BASICS handle it correctly as the value of θ approaches 0. It should be obvious that, as θ approaches zero, the value of SIN θ will also approach zero, since SIN θ is zero when θ is zero. In other words, as the angle gets nearer and nearer to zero, so the arc on the circumference that defines θ comes closer and closer to zero, and the point at which the hypotenuse intersects the circle gets closer and closer to 0 on the y-axis. Unfortunately, the precision of BASIC is limited. In other words, BASIC can only handle very large values up to a certain value and very small values down to a certain value. If θ is very small (say 1.0E-36, i.e. 1×10 to the power of minus 36), then BASIC may not be able to cope and will simply return a value of 0 for the sine of such numbers. Before using the sine function, try testing your BASIC using the following small program:

Sine Underflow Or Roundoff Errors

```
1 REM TEST FOR SIN FUNCTION ROUNDOFF OR UNDERFLOW ERRORS
10 LET X = 10/6
20 PRINT "ITERATION"," VAL OF X"," VAL OF SIN(X)"
30 FOR I = 1 TO 40
40 LET X = X / 10
50 PRINT I,X,SIN(X)
60 NEXT I
70 END
```

ITERATION	VAL OF X	VAL OF SIN(X)
1	.166667	.165896
2	.0166667	.0166659
3	1.66667E-03	1.66667E-03
4	1.66667E-04	1.66667E-04
5	1.66667E-05	1.66667E-05
6	1.66667E-06	1.66667E-06
7	1.66667E-07	1.66667E-07
8	1.66667E-08	1.66667E-08
9	1.66667E-09	1.66667E-09
10	1.66667E-10	1.66667E-10
11	1.66667E-11	1.66667E-11
12	1.66667E-12	1.66667E-12
13	1.66667E-13	1.66667E-13
14	1.66667E-14	1.66667E-14
15	1.66667E-15	1.66667E-15
16	1.66667E-16	1.66667E-16
17	1.66667E-17	1.66667E-17
18	1.66667E-18	1.66667E-18
19	1.66667E-19	1.66667E-19
20	1.66667E-20	1.66667E-20
21	1.66667E-21	1.66667E-21
22	1.66667E-22	1.66667E-22
23	1.66667E-23	1.66667E-23
24	1.66667E-24	1.66667E-24
25	1.66667E-25	1.66667E-25
26	1.66667E-26	1.66667E-26
27	1.66667E-27	1.66667E-27
28	1.66667E-28	1.66667E-28
29	1.66667E-29	1.66667E-29
30	1.66667E-30	1.66667E-30
31	1.66667E-31	1.66667E-31
32	1.66667E-32	1.66667E-32
33	1.66667E-33	1.66667E-33
34	1.66667E-34	1.66667E-34
35	1.66667E-35	1.66667E-35
36	1.66667E-36	1.66667E-36
37	1.66667E-37	1.66667E-37
38	1.66667E-38	1.66667E-38
39	0	0
40	0	0

A run of this program using Microsoft's MBASIC is given. This particular BASIC interpreter handles the SIN of small numbers quite well and doesn't cause trouble until the value of θ is less than 1.0E-38 (a decimal point followed by 37 zeros).

The program given depends on an adequate dynamic range in BASIC's handling of floating point arithmetical operations. It is well to remember that before you can use any mathematical operation in BASIC with confidence, you need to be aware of the range of numbers it can handle accurately.

Remember that a variable name alone, such as X or TREND will automatically be *single precision* (i.e. capable of storing no more than seven digits). Alternatively, variables can be specified as, or changed to, single precision by appending an exclamation mark, as in X! or TREND!. *Double precision* variables (which can store 17 digits) are specified by appending a hash sign, as in X# or TREND#. *Integer* variables (which can store only whole numbers) are specified in many versions of BASIC by appending a per cent sign, as in X% or TREND%.

We end this article with a short program that lets you test how many digits can be stored in a variable in your version of BASIC, together with a print-out of the program when run using Microsoft BASIC. There are two versions, one for testing small numbers and one for large ones. The print-out for small numbers shows that as the numbers become very small (less that 3.3 × 1OE-38) BASIC rounds the numbers off to zero. For large numbers (greater than 3.3 × 1OE37) an overflow occurs and the results are unreliable. If you need to work with very large or very small numbers you may need to write special arithmetic routines to overcome these limitations.

Small Numbers In BASIC

```
1 REM  TESTS HANDLING OF SMALL NUMBERS IN BASIC
10 LET X# =.000033333333333#
20 PRINT "ITERATION"," DBL PREC"," ","SNGL PREC"
30 PRINT
40 FOR I = 1 TO 40
50 LET X# = X# / 10
60 LET X! = X#
70 PRINT I,X#,X!
80 NEXT I
90 END
```

ITERATION	DBL PREC	SNGL PREC
1	.0000033333333333	3.33333E-06
2	.0000003333333333	3.33333E-07
3	3.333333333D-08	3.33333E-08
4	3.333333333D-09	3.33333E-09
5	3.333333333D-10	3.33333E-10
6	3.333333333D-11	3.33333E-11
7	3.333333333D-12	3.33333E-12
8	3.333333333D-13	3.33333E-13
9	3.333333333D-14	3.33333E-14
10	3.333333333D-15	3.33333E-15
11	3.333333333D-16	3.33333E-16
12	3.333333333D-17	3.33333E-17
13	3.333333333D-18	3.33333E-18
14	3.333333333D-19	3.33333E-19
15	3.333333333D-20	3.33333E-20
16	3.333333333D-21	3.33333E-21
17	3.333333333D-22	3.33333E-22
18	3.333333333D-23	3.33333E-23
19	3.333333333D-24	3.33333E-24
20	3.333333333D-25	3.33333E-25
21	3.333333333D-26	3.33333E-26
22	3.333333333D-27	3.33333E-27
23	3.333333333D-28	3.33333E-28
24	3.333333333D-29	3.33333E-29
25	3.333333333D-30	3.33333E-30
26	3.333333333D-31	3.33333E-31
27	3.333333333D-32	3.33333E-32
28	3.333333333D-33	3.33333E-33
29	3.333333333D-34	3.33333E-34
30	3.333333333D-35	3.33333E-35
31	3.333333333D-36	3.33333E-36
32	3.333333333D-37	3.33333E-37
33	3.333333333D-38	3.33334E-38
34	0	0
35	0	0
36	0	0
37	0	0
38	0	0
39	0	0
40	0	0

Large Numbers In BASIC

```
 1 REM  TESTS HANDLING OF LARGE NUMBERS IN BASIC
10 LET X# = 3.3333333333334#
20 PRINT "ITERATION"," DBL PREC"," ","SNGL PREC"
30 PRINT
40 FOR I = 1 TO 40
50 LET X# = X# * 10
60 LET X! = X#
70 PRINT I,X#,X!
80 NEXT I
90 END
```

ITERATION	DBL PREC	SNGL PREC
1	33.333333333334	33.3333
2	333.33333333334	333.333
3	3333.3333333334	3333.33
4	33333.333333334	33333.3
5	333333.33333334	333333
6	3333333.3333334	3.33333E+06
7	33333333.333334	3.33333E+07
8	333333333.33334	3.33333E+08
9	3333333333.3334	3.33333E+09
10	33333333333.334	3.33333E+10
11	333333333333.34	3.33333E+11
12	3333333333333.4	3.33333E+12
13	33333333333333	3.33333E+13
14	333333333333334	3.33333E+14
15	3333333333333334	3.33333E+15
16	3.3333333333333334D+16	3.33333E+16
17	3.3333333333333334D+17	3.33333E+17
18	3.3333333333333334D+18	3.33333E+18
19	3.3333333333333334D+19	3.33333E+19
20	3.3333333333333334D+20	3.33333E+20
21	3.3333333333333334D+21	3.33333E+21
22	3.3333333333333334D+22	3.33333E+22
23	3.3333333333333334D+23	3.33333E+23
24	3.3333333333333334D+24	3.33333E+24
25	3.3333333333333334D+25	3.33333E+25
26	3.3333333333333334D+26	3.33333E+26
27	3.3333333333333334D+27	3.33333E+27
28	3.3333333333333334D+28	3.33333E+28
29	3.3333333333333334D+29	3.33333E+29
30	3.3333333333333334D+30	3.33333E+30
31	3.3333333333333335D+31	3.33333E+31
32	3.3333333333333335D+32	3.33333E+32
33	3.3333333333333334D+33	3.33333E+33
34	3.3333333333333335D+34	3.33333E+34
35	3.3333333333333335D+35	3.33333E+35
36	3.3333333333333335D+36	3.33333E+36
37	3.3333333333333335D+37	3.33333E+37
38	1.7014118346046930+38	1.70141E+38
39	1.7014118346046930+38	1.70141E+38
40	1.7014118346046930+38	1.70141E+38

STARTING FLAG

Having already used the add instruction in previous instalments of the course, we now begin to examine its implications in terms of methods of arithmetic, and the system architecture needed to support them. Here, we look more closely at the processor status register and its part in addition — in particular the role of the carry flag.

The add instruction in both Z80 and 6502 Assembly language is ADC — meaning 'Add With Carry' — a mnemonic of great importance for Assembly language programming. The concept of a 'carry' bit is of particular significance. Let's consider the addition of two hex numbers in the accumulator:

```
   A7      =     10100111
+  3E      =   + 00111110
-----------------------
   E5      =     11100101
```

Since the accumulator is an eight-bit register, both the numbers to be added and the sum itself must be in the range $00 to $FF (as they are here) or else they will not fit into the accumulator. Does this mean, therefore, that we are restricted to additions in which the sum is less than $100? Consider another addition in the accumulator, one which violates this restriction:

```
   FF      =     11111111
+  FF      =   + 11111111
-----------------------
  1FE      =    111111110
```

This shows the addition of the largest possible single-byte numbers, and seems to be an illegal addition. It requires a nine-bit accumulator. The solution to this dilemma is suggested in the statement of the problem — we need only an extra bit on the accumulator to contain the largest number that can be generated by the addition of single-byte numbers. That extra bit is required only in the sum, not in the addition operands, and it is required only when there is a 'carry' from the most significant bit of the accumulator.

PROCESSOR STATUS REGISTER

The extra bit is therefore known as the *carry bit*, and it is located in the eight-bit register associated with the accumulator known as the *processor status register* (PSR). This important register is connected to the accumulator and the ALU in such a way that individual bits of the PSR are set or cleared following any accumulator operation, depending on the results of that operation. The contents of the process status register can be regarded as a simple number, but it is usually more informative to treat it as an eight-element array of binary flags, whose individual states show the particular effects of the last operation (a *flag* is any variable whose value indicates the state or truth-value of some condition, rather than being an absolute value. A flag variable usually has only two states or conditions: up or down, on or off, 0 or 1).

When any operation is performed on the accumulator that causes a carry out of the eighth bit, then the carry flag of the PSR will be set automatically to 1; an operation that does not cause a carry will reset (set to 0) the carry flag. This applies only to those operations that might legitimately cause a carry. Some operations, such as loading to or storing from the accumulator, do not affect the carry flag. Whenever we investigate a new Assembly language instruction in the course from now on, we shall want to know which of the PSR (or flag register) bits it affects. Naturally, we shall need to know more about the other PSR flags, but let's finish our discussion of the carry flag first.

In general, when adding two single-byte numbers we won't know in advance what they will be, so we have to be prepared for the sum of such an addition to exceed $FF; usually this will mean reserving two bytes of RAM to hold the result of an addition. Consider, again, the previous addition examples:

Hex Numbers		Carry Flag	Binary Numbers
A7	=		10100111
+ 3E	=		+ 00111110
00E5	=	0	11100101
		No Carry	
FF	=		11111111
+ FF	=		+ 11111111
01FE	=	1	111111110
		Carry	

The result of the addition is represented in both examples as a two-byte number. In the first case, the carry flag is reset to 0 because there is no carry out from the eighth bit of the sum (the two-byte result is $00E5, of which the hi-byte is $00). In the second case, however, there is a carry out from the eighth bit, so the carry flag is set, and the hi-byte of the result is $01.

To be sure of getting the correct result of an addition, therefore, we must store the accumulator contents in the lo-byte of the two-

byte location, then store the carry flag as the hi-byte of that location. There is no single instruction for storing the carry flag, but the ADC op-code was formulated with precisely this operation in mind: ADC actually means 'add the instruction operand to the current contents of the carry flag, then add that result to the contents of the accumulator'. Addition is thus a two-stage process, in the first of which the current state of the carry flag is used, while in the second stage the state of the carry flag is updated.

This means, then, that before beginning an addition, we must consider the current state of the carry flag, since it will be added into the addition sum proper: hence the two unexplained instructions in previous instalments, CLC and AND A. The former, a 6502 instruction, means 'clear the carry flag', and does exactly that. The Z80 version, AND A, means 'logically AND the accumulator with itself'. While not designed solely to reset the carry flag it does have that effect and doesn't affect anything else, so is often used as a Z80 equivalent of the 6502's CLC.

Having cleared the carry flag before starting an addition, therefore, we must store its contents afterwards. This is achieved by adding the immediate value $00 to the hi-byte of the result. This won't affect the byte if the carry flag is clear, but will add 1 to it if the carry flag is set.

All of what we have said in this instalment leads to the first method for single-byte arithmetic:

1) Clear the carry flag
2) Load the accumulator with one number
3) Add in the second number
4) Store the contents of the accumulator in the lo-byte of a two-byte location
5) Load the accumulator with the contents of the hi-byte
6) Add in the immediate value $00
7) Store the contents of the accumulator in the hi-byte

When this procedure is turned into Assembly language we get:

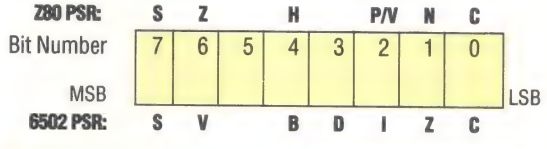
COMMON TO BOTH PROCESSORS		
Label	**Directive**	**Operand**
BYTE1	EQU	$FF
BYTE2	EQU	$FF
LOBYTE	EQU	$A000
HIBYTE	EQU	$A001
	ORG	$A020

Z80		6502	
Op-code	**Operand**	**Op-code**	**Operand**
LD	A,$00	LDA	#$00
LD	(HIBYTE),A	STA	HIBYTE
AND	A	CLC	
LD	A,BYTE1	LDA	#BYTE1
ADC	A,BYTE2	ADC	#BYTE2
LD	(LOBYTE),A	STA	LOBYTE
LD	A,(HIBYTE)	LDA	HIBYTE
ADC	A,$00	ADC	#$00
LD	(HIBYTE),A	STA	HIBYTE
RET		RTS	

Remember that the values given for LOBYTE, HIBYTE and ORG are for example only — you must choose values appropriate to the machine that you use. Notice that the first two instructions of the program load $00 into HIBYTE, so that it's not corrupted by random data. We don't have to clear LOBYTE in the same way because its starting contents are overwritten with the lo-byte of the result.

It is worth remarking again about the differences of approach between Z80 and 6502 Assembly language as seen in the example. The 6502 code reads quite simply once you're used to it — the mnemonics themselves and the use of '#' to signal immediate data make the meaning of each instruction clear. The Z80 version is less straightforward because the LD mnemonic is used for all data transfers whether into or out of the accumulator. Also, there is no '#' symbol to signal immediate data, only the absence of brackets around the operand indicate this. Thus LD A,BYTE1 means 'load the accumulator with the immediate data BYTE1'; whereas LD A,(HIBYTE) means 'load the accumulator from the address HIBYTE'. In the full Assembly language listing there is no ambiguity in the meaning of such instructions, since the hex value of the op-code uniquely identifies the instruction. This may seem to beg the question, however — the op-code may be unique, but if there is a choice of unique op-codes, how does the assembler (or the person doing the assembly) choose between them? The answer lies in the Addressing Mode, which will be the topic of the next instalment.

Finally, we should take note that the processor status register contains other flags as well as the carry flag, which we'll examine briefly now, and return to in detail later in the course:



PSR BIT	Z80
7	(S)=SIGN
6	(Z)=ZERO
5	unused
4	(H)=HALF-CARRY
3	unused
2	(P/V)=PARITY/OVERFLOW
1	(N)=SUBTRACT
0	(C)=CARRY

6502	PSR BIT
(S)=SIGN	7
(V)=OVERFLOW	6
unused	5
(B)=BREAK	4
(D)=BCD MODE	3
(I)=INTERRUPT	2
(Z)=ZERO	1
(C)=CARRY	0

For our present purposes the important flags are the carry, sign and zero flags. We have seen that after an addition the carry flag holds the value of the carry out of the eighth bit of the accumulator. The sign flag is always a copy of the eighth bit (bit 7) of the accumulator, and the zero flag is set to 1 if the accumulator contents are zero, and reset to 0 if the contents are non-zero.

Answers To Assembly Exercise On Page 158

1) The assembled programs are given in the box on the right.
Notice that the symbols BYTE1 and BYTE2 are used as both immediate symbolic data, and as symbolic addresses. When they are used as the latter, however, they need to be assembled in two-byte form.

2) The 'return from subroutine' instruction is missing from the end of both programs. In the 6502 version, the completed code would need to include the following line:

```
A00D        60          RTS
```

and the Z80 version needs this line:

```
A00D        C9          RET
```

3) The value $45 is first loaded into the accumulator register as immediate data, and then $45 is added on top of it, so that the accumulator contains the value $8A. This accumulated total is then stored in RAM at address $0045. The value $38 is then added into the accumulator as immediate data, so that the accumulator now contains the value $C2 ($45+$45+$38). This total is finally stored in RAM at location $0038.

4) 'Immediate data' is data that is actually stored in the instruction. In the instructions we gave in the exercise programs (such as LDA #$9C and LD A,$E4) the values $9C and $E4 are the data to be loaded into the accumulator. They are stored in the instructions of which they are operands, and comprise the contents of the byte immediately following the op-code. If data is not available, then it must be stored in some other part of memory, and be referred to by its address rather than its value.

5) The value of BYTE1 is given as $45, which properly written gives memory location $0045. Clearly, this address is on page zero of memory.

Exercise

We may wish to examine the contents of the processor status register (PSR), and it will be convenient to display this number as a binary rather than a hex byte. We include here the Spectrum version of a 'decimal-to-binary conversion subroutine'. This exercise asks you to patch this into the Monitor program on page 118.

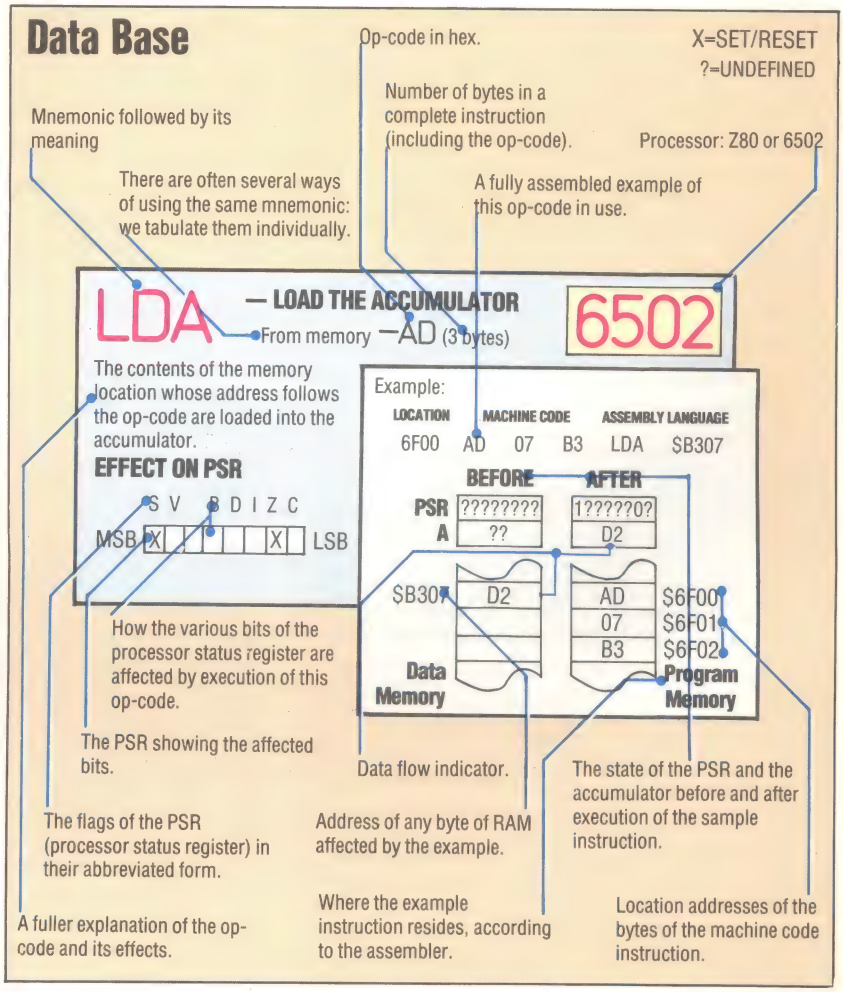
```
7000 REM******BINARY BYTE S/R*******
7001 REM*CONVERTS A NUMBER N (<256)*
7002 REM*TO AN 8-CHARACTER BINARY   *
7003 REM*REPRESENTATION IN B$       *
7010 B$=""
7020 FOR D=8 TO 1 STEP-1
7030 LET N1=INT(N/2)
7040 LET R=N-2*N1
7050 LET B$=STR$(R)+B$
7060 LET N=N1
7070 NEXT D
7080 RETURN
```

Basic Flavours

On the Commodore 64, change line 7050 in the subroutine to:

```
7050 B$=MID$(STR$(R),2)+B$
```

Location Address	Machine Code	Assembly Language		
6502				
0000		START	EQU	$A000
0000		BYTE1	EQU	$45
0000		BYTE2	EQU	$38
0000			ORG	START
A000	A9 45		LDA	#BYTE1
A002	18		CLC	
A003	69 45		ADC	#BYTE1
A005	8D 45 00		STA	BYTE1
A008	69 38		ADC	#BYTE2
A00A	8D 38 00		STA	BYTE2
Z80				
0000		START	EQU	$A000
0000		BYTE1	EQU	$45
0000		BYTE2	EQU	$38
0000			ORG	START
A000	3E 45		LD	A,BYTE1
A002	A7		AND	A
A003	CE 45		ADC	A,BYTE1
A005	32 45 00		LD	(BYTE1),A
A008	CE 38		ADC	A,BYTE2
A00A	32 38 00		LD	(BYTE2),A


Data Base
Op-code in hex.
X=SET/RESET
?=UNDEFINED
Mnemonic followed by its meaning
Number of bytes in a complete instruction (including the op-code).
Processor: Z80 or 6502
There are often several ways of using the same mnemonic: we tabulate them individually.
A fully assembled example of this op-code in use.
LDA
— LOAD THE ACCUMULATOR
From memory —AD (3 bytes)
6502
The contents of the memory location whose address follows the op-code are loaded into the accumulator.
EFFECT ON PSR
S V B D I Z C
MSB X LSB X
Example:
LOCATION MACHINE CODE ASSEMBLY LANGUAGE
6F00 AD 07 B3 LDA $B307
BEFORE AFTER
PSR ???????? 1??????0?
A ?? D2
$B307 D2
AD $6F00
07 $6F01
B3 $6F02
Data Memory
Program Memory
How the various bits of the processor status register are affected by execution of this op-code.
The PSR showing the affected bits.
Data flow indicator.
The state of the PSR and the accumulator before and after execution of the sample instruction.
The flags of the PSR (processor status register) in their abbreviated form.
Address of any byte of RAM affected by the example.
A fuller explanation of the op-code and its effects.
Where the example instruction resides, according to the assembler.
Location addresses of the bytes of the machine code instruction.

LDA – LOAD THE ACCUMULATOR — 6502

Immediate –A9 (2 bytes)

The contents of the byte following the op-code are loaded into the accumlator.

EFFECT ON PSR

S V B D I Z C

MSB [X][ ][ ][ ][ ][X][ ] LSB

Example:

LOCATION	MACHINE CODE	ASSEMBLY LANGUAGE
A000	A9 3F	LDA #$3F

	BEFORE	AFTER
PSR	????????	0??????0
A	??	3F

	Data Memory	Program Memory	
		A9	$A000
		3F	$A001

LD A, — LOAD THE ACCUMULATOR — Z80

Immediate –3E (2 bytes)

The contents of the byte following the op-code are loaded into the accumulator.

EFFECT ON PSR

S Z H V N C

MSB [ ][ ][ ][ ][ ][ ][ ] LSB

NO EFFECT

Example:

LOCATION	MACHINE CODE	ASSEMBLY LANGUAGE
A000	3E 9B	LD A,$9B

	BEFORE	AFTER
PSR	????????	????????
A	??	9B

	Data Memory	Program Memory	
		3E	$A000
		9B	$A001

LDA – LOAD THE ACCUMULATOR — 6502

From memory –AD (3 bytes)

The contents of the memory location whose address follows the op-code are loaded into the accumulator.

EFFECT ON PSR

S V B D I Z C

MSB [X][ ][ ][ ][ ][X][ ] LSB

Example:

LOCATION	MACHINE CODE	ASSEMBLY LANGUAGE
6F00	AD 07 B3	LDA $B307

	BEFORE	AFTER
PSR	????????	1??????0
A	??	D2

	Data Memory	Program Memory	
$B307	D2	AD	$6F00
		07	$6F01
		B3	$6F02

LD A, — LOAD THE ACCUMULATOR — Z80

From memory –3A (3 bytes)

The contents of the memory location whose address follows the op-code are loaded into the accumulator.

EFFECT ON PSR

S Z H V N C

MSB [ ][ ][ ][ ][ ][ ][ ] LSB

NO EFFECT

Example:

LOCATION	MACHINE CODE	ASSEMBLY LANGUAGE
6F00	3A E9 F4	LD A,($F4E9)

	BEFORE	AFTER
PSR	????????	????????
A	??	8C

	Data Memory	Program Memory	
$F4E9	8C	3A	$6F00
		E9	$6F01
		F4	$6F02

STA – STORE THE ACCUMULATOR — 6502

To memory –8D (3 bytes)

The contents of the accumulator are loaded into the memory location whose address follows the op-code.

EFFECT ON PSR

S V B D I Z C

MSB [ ][ ][ ][ ][ ][ ][ ] LSB

NO EFFECT

Example:

LOCATION	MACHINE CODE	ASSEMBLY LANGUAGE
E532	8D A2 65	STA $65A2

	BEFORE	AFTER
PSR	????????	????????
A	68	68

	Data Memory	Program Memory	
$65A2	68	8D	$E532
		A2	$E533
		65	$E534

LD( ),A — LOAD THE ACCUMULATOR — Z80

To memory –32 (3 bytes)

The contents of the accumulator are loaded into the memory location whose address follows the op-code.

EFFECT ON PSR

S Z H V N C

MSB [ ][ ][ ][ ][ ][ ][ ] LSB

NO EFFECT

Example:

LOCATION	MACHINE CODE	ASSEMBLY LANGUAGE
E532	32 E9 F4	LD ($F4E9),A

	BEFORE	AFTER
PSR	????????	????????
A	4B	4B

	Data Memory	Program Memory	
$F4E9	4B	32	$E532
		E9	$E533
		F4	$E534

ADC – ADD WITH CARRY — 6502

Immediate –69 (2 bytes)

The carry flag plus the contents of the byte following the op-code are added to the contents of the accumulator.

EFFECT ON PSR

S V B D I Z C

MSB [X][X][ ][ ][ ][X][X] LSB

Example:

LOCATION	MACHINE CODE	ASSEMBLY LANGUAGE
840B	69 A2	ADC #$A2

	BEFORE	AFTER
PSR	00????01	11????00
A	16	B9

	Data Memory	Program Memory	
$840B	69	A2	$840C

ADC A, — ADD WITH CARRY — Z80

Immediate –69 (2 bytes)

To the contents of the accumulator are added the carry flag plus the contents of the byte following the op-code.

EFFECT ON PSR

S Z H V N C

MSB [X][X][X][X][X][X][ ] LSB

Example:

LOCATION	MACHINE CODE	ASSEMBLY LANGUAGE
840B	CE B9	ADC A,$B9

	BEFORE	AFTER
PSR	00?0?001	10?0?1?0
A	24	DE

	Data Memory	Program Memory	
		CE	$840B
		B9	$840C

GETTING IT TAPED

Virgin Games is a subsidiary of the highly successful independent record company, Virgin Records. The connections between promoting popular music and computer programs are significant: software 'pop charts' are becoming as important as the Top Forty.

When the market for home computer software was an unknown quantity, back in the early 1970s, it afforded many opportunities for youthful entrepreneurs to cash in on the demand for cassette software. Anyone who could write amusing games programs in BASIC could get hold of a high-speed cassette-to-cassette audio dubbing machine and sell mail-order through the small ads.

Today, things are not so simple. No matter how good a programmer you are, you must possess originality and creativity in order to make the grade. Different software houses originate their product in different ways. Imagine Software for instance, (see page 79) has many in-house-programmers coding their bosses' creative ideas.

Nick Alexander, the 28-year-old managing director of Virgin Games, says: 'Often the better the programmer, the fewer the ideas they have. To be a good programmer you need to be very logical, very methodical, very diligent, and those tend not to be the qualities of the creative individual'. For this reason, he has chosen to restrict Virgin's in-house programmers to a minimum. The plan is to provide a technical and creative service to correct the deficiencies in the many programs that they receive every week from young hopefuls. Capable programmers are helped in developing ideas; and the creative people get help with coding.

The rewards of being published by Virgin may appear less than those from other companies. A game that Virgin publishes earns an advance of between £1,000 and £3,000 for the author against 7.5 per cent royalties on the net price. Contrast this with the 25 per cent royalties that many other software publishers claim to offer. But Alexander argues that because nearly a quarter of the net revenue of any game is ploughed back into promoting it, sales (and the author's eventual reward) are subsequently much greater.

Promoting products is, of course, an activity that Virgin knows a lot about. Virgin's name was established through its successful ventures in the music business and the techniques that 31-year-old Virgin boss Richard Branson learned in that field have been applied to its software offshoot. Games writers are promoted as stars in their own right — cassette inlays not only credit the author by name, but also feature a picture and thumbnail biography. Virgin Games, which started in February 1983 by advertising for games in the home computer magazines, received 500 initial submissions. Now it has 46 titles for eight home computers on its list. Its best sales are for the Spectrum, with the BBC Micro in second place and the Commodore 64 not far behind.

Virgin's hottest new writer is Martin Wheeler, who is 15 and has just written two new games for the Spectrum, 'Dr Franky and the Monsters' and 'Sorcerers'. Wheeler has assembled the programs in machine code, and developed some impressive graphics for them. Alexander sees a similarity between the home computing scene of today and the music business of a decade ago and believes that computing is on the way to displacing music as the favoured leisure activity of the young.

GET MACHINE CODE TAPED WITH CHAMP

* CHAMP: A uniquely comprehensive aid to machine code programming
* Specially commissioned to accompany the *Home Computer Advanced Course* series on machine code
* Suitable for the BBC Model B, Commodore 64 and 48K Sinclair Spectrum
* Pre-recorded on tape cassette FREE with Issue 11 of *The Home Computer Advanced Course*

To unlock the full potential of your computer you have to talk to it in its own language — machine code. The *Home Computer Advanced Course* is teaching you to do just that in its unrivalled series on machine code programming. And now you can bring the computer itself to your aid. With the pre-recorded software that is being given away with Issue 11, you can tackle ambitious programs — thanks to these features, rarely found together in one package:

The Editor enables you to enter your programs, written in Assembly language mnemonics, from the keyboard, or to load them from tape. You can then modify the program freely, with on-entry syntax checking.

The Assembler translates your Assembly language program into machine code — saving you hours of tricky, error-prone work

The Monitor displays contents of memory, together with hex and ASCII equivalents, and enables you to move blocks of memory, search for specified strings, and modify the memory contents directly

The Disassembler translates the machine code version of the program back into Assembly language mnemonics for your scrutiny

The Debug Facility enables you to run the program step by step while displaying the contents of the microprocessor's registers and modifying them as necessary

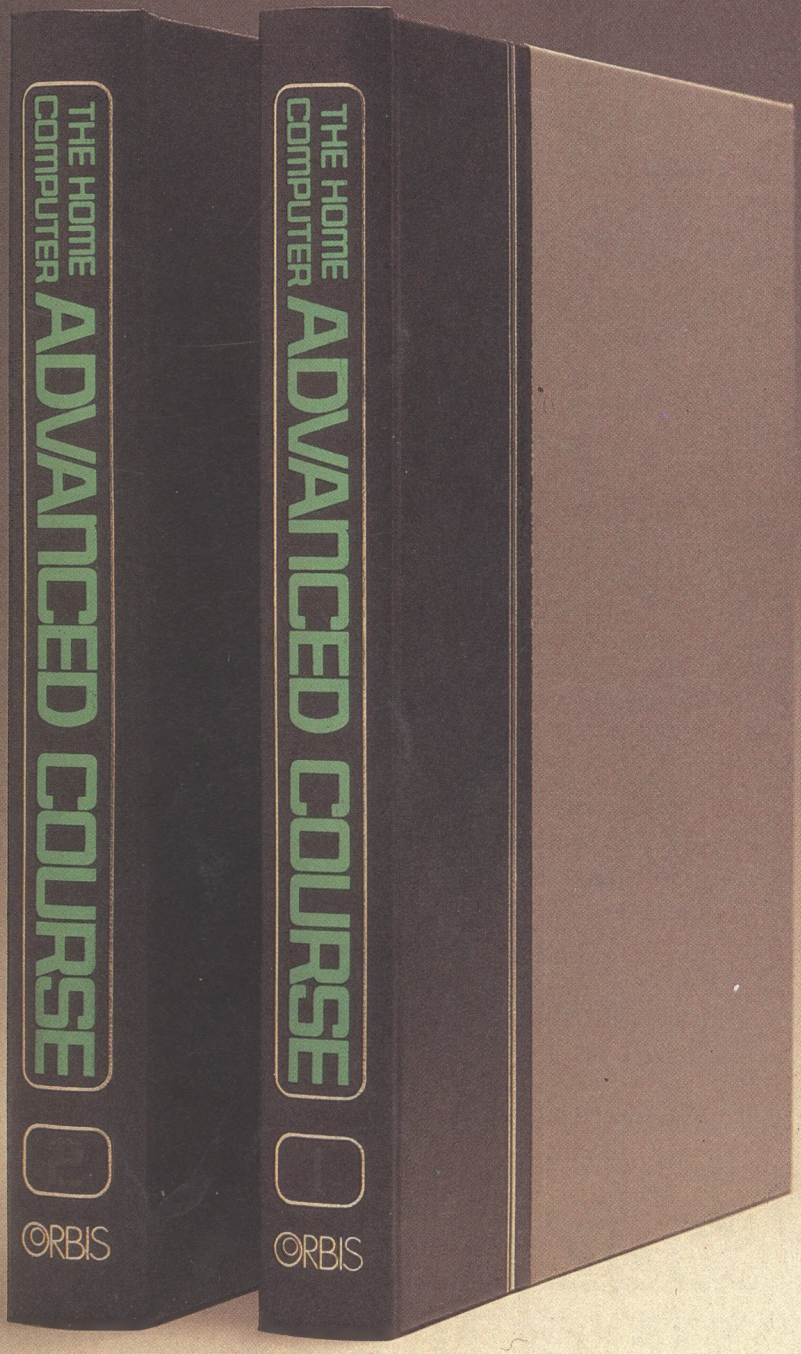
Normally you'd have to buy several different software packages to obtain all these facilities — and you'd have to spend pounds. They're *free* with Issue 11 of *The Home Computer Advanced Course*. This suite of advanced programs was specially commissioned from a leading software company, Personal Software Services, to be a valuable programming aid for readers. With its aid you'll learn how to make your micro do things that will amaze you: seemingly instantaneous program execution and dazzling, lightning-fast graphics animation.

(CHAMP — Comprehensive Home Computer User's Assembler/ Monitor Package)

GET MACHINE CODE TAPED WITH ISSUE 11 OF THE HOME COMPUTER ADVANCED COURSE

THE HOME COMPUTER ADVANCED COURSE

WE HAVE DESIGNED BINDERS SPECIALLY TO KEEP YOUR COPIES OF THE 'ADVANCED COURSE' IN GOOD ORDER.



All you have to do is complete the reply-paid order form opposite – tick the box and post the card today – **no stamp necessary!**

By choosing a standing order, you will be sent the first volume free along with the second binder for £3.95. The invoice for this amount will be with the binder. We will then send you your binders every twelve weeks – as you need them.

Important: This offer is open only whilst stocks last and only one free binder may be sent to each purchaser who places a Standing Order. Please allow 28 days for delivery.

Overseas readers: This free binder offer applies to readers in the UK, Eire and Australia only. Readers in Australia should complete the special loose insert in issue 1 and see additional binder information on the inside front cover. Readers in New Zealand and South Africa and some other countries can obtain binders now. For details please see inside the front cover. Binders may be subject to import duty and/or local tax.

The Orbis Guarantee: If you are not entirely satisfied you may return the binder(s) to us within 14 days and cancel your Standing Order. You are then under no obligation to pay and no further binders will be sent except upon request.

PLACE A STANDING ORDER TODAY.